AGARD-AR-292

AGARD-AR-292

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT

7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD ADVISORY REPORT 292

Automated Software Generation Approaches for the Design and Development of Guidance and Control Systems Software

(Les Différentes Approches "Génération" pour la Conception et le Développement de Logiciels de Guidage et de Pilotage)

This Advisory Report has been prepared as a summary of the deliberations of Working Group 10 of the Guidance and Control Panel of AGARD.

NORTH ATLANTIC TREATY ORGANIZATION

Published March 1992

Distribution and Availability on Back Cover

AGARD-AR-292

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT

7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD ADVISORY REPORT 292

Automated Software Generation Approaches for the Design and Development of Guidance and Control Systems Software

(Les Différentes Approches "Génération" pour la Conception et le Développement de Logiciels de Guidage et de Pilotage)

Edited by

Dr Edwin B. Stear
Corporate Vice President, Technical Assessment, The Boeing Company
Post Office Box 3707 M/S 13-43, Seattle, WA 98124-2207, United States

and

Prof. John T. Shepherd
Research Director, GEC-Marconi Limited, Elstree Way,
Borehamwood, Hertfordshire WD6 1RX, United Kingdom

This Advisory Report has been prepared as a summary of the deliberations of Working Group 10 of the Guidance and Control Panel of AGARD.

North Atlantic Treaty Organization
Organisation du Traité de l'Atlantique Nord

The Mission of AGARD

According to its Charter, the mission of AGARD is to bring together the leading personalities of the NATO nations in the fields of science and technology relating to aerospace for the following purposes:

— Recommending effective ways for the member nations to use their research and development capabilities for the common benefit of the NATO community;

— Providing scientific and technical advice and assistance to the Military Committee in the field of aerospace research and development (with particular regard to its military application);

— Continuously stimulating advances in the aerospace sciences relevant to strengthening the common defence posture;

— Improving the co-operation among member nations in aerospace research and development;

— Exchange of scientific and technical information;

— Providing assistance to member nations for the purpose of increasing their scientific and technical potential;

— Rendering scientific and technical assistance, as requested, to other NATO bodies and to member nations in connection with research and development problems in the aerospace field.

The highest authority within AGARD is the National Delegates Board consisting of officially appointed senior representatives from each member nation. The mission of AGARD is carried out through the Panels which are composed of experts appointed by the National Delegates, the Consultant and Exchange Programme and the Aerospace Applications Studies Programme. The results of AGARD work are reported to the member nations and the NATO Authorities through the AGARD series of publications of which this is one.

Participation in AGARD activities is by invitation only and is normally limited to citizens of the NATO nations.

The content of this publication has been reproduced
directly from material supplied by AGARD or the authors.

Published March 1992

ISBN 92-835-0663-4

Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

Preface

Recognizing that there is an urgent need for greatly improved design methods and tools to support the development of guidance and control software, the AGARD Guidance and Control Panel began investigating the state of software design and software engineering environments for navigation, guidance, and control systems. AGARD Guidance and Control Working Group 08 determined and reported in 1986 that while current tools being developed at that time would significantly improve the cost and integrity of software, this improved software would still be overly costly. It was apparent that even greater improvements in methods and tools were needed if software development were not to become a major bottleneck in the development of future military systems.

Realizing that a new class of software design support tools (called software generators) is being developed to solve similar problems in the business domain, the AGARD Guidance and Control Panel formed Working Group 10 to investigate this new class of generators for applicability in guidance and control systems. Working Group 10 defined and investigated four approaches to software generation: reusable software modules, expert systems, program transformation techniques, and fourth generation languages. This report contains the findings and conclusions of Working Group 10.

The Working Group was formed in May 1987 and met at six-month intervals through October 1990. Briefings on the findings of this study were given by members of the group at final editing of the 52nd Guidance and Control Panel Symposium on "Software for Guidance and Control", held in Thessaloniki, Greece on May 10, 1991. Final editing of the report took place from May 1991 to November 1991.

The Working Group was composed of members from France, Germany, the Netherlands, the United Kingdom, and the United States, all of whom are expert in either guidance and control systems design or software design or both. This report represents the consensus view of the group, but it should not be construed as representing the views or policies of any of the nations, organizations, or individuals represented on the Working Group.

Préface

Le Panel AGARD du Guidage et du Pilotage, reconnaissant le besoin indispensable d'amélioration des méthodes de conception et des outils supports pour le développement des logiciels de guidage, et de pilotage, a entrepris l'examen des ateliers logiciels dédiés aux problèmes de navigation, de guidage et de pilotage. Dans son rapport en 1986, le groupe de travail GCP 08 a montré que, bien que les outils actuellement disponibles améliorent sensiblement les coûts et la productivité, les logiciels ainsi produits demeurent encore trop chers. Il a paru évident au groupe qu'il fallait encore plus d'améliorations pour que le développement des logiciels ne soit pas le goulot d'étranglement du développement des futurs systèmes militaires.

Conscient du fait que de nouvelles méthodes et leurs outils correspondants (appelés "Générateurs de Logiciels") étaient en cours de développement pour résoudre des problèmes similaires dans le monde de la gestion, le Panel AGARD du Guidage et du Pilotage a créé le groupe de travail WG 10 (successeur du WG 08) afin d'évaluer les possibilités d'application de cette nouvelle classe de techniques aux problèmes de guidage et de pilotage. Le groupe de travail a défini et examiné quatre approches complémentaires pour la génération de logiciels: les modules logiciels réutilisables, les systèmes experts, les techniques de transformation des programmes et les langages de quatrième génération. Ce rapport contient les constatations et les conclusions du groupe de travail No.10.

Le groupe de travail fut créé au mois de mai 1987 et il s'est réuni tous les six mois jusqu'au mois d'octobre 1990. Les membres du groupe ont donné des briefings sur les conclusions définitives de l'étude lors du 52ème symposium du Panel GCP "Les logiciels du guidage et du pilotage" à Thessalonique, en Grèce, le 10 mai 1991. La mise en forme définitive du rapport s'est effectuée de mai à novembre 1991.

Le groupe de travail étati composé de membres de la France, de l'Allemagne, des Pays-Bas, du Royaume-Uni et des Etats-Unis, tous experts dans le domaine de la conception des systèmes de guidage et de pilotage et/ou la création de logiciels. Ce rapport exprime le point de vue du groupe et ne reflete ni les opinions ni la politique des pays, des organisations ou des personnes ayant fait partie du groupe de travail No.10

Guidance and Control Panel
Working Group 10

France

Dr Patrick DeBondeli
Conception et Réalisation d'Applications Automatisées

Mr Michael Lemoine
Département Etudes et Recherche en Informatique/CERT

Germany

Mr B.V. Guretzky
MBB Deutsche Aerospace

Mr H. Neumann
MBB Deutsche Aerospace

Mr Hansjoerg F. Roschmann
TST Deutsche Aerospace

Mr E.D. Holzbaur
TST Deutsche Aerospace

Mr W. Mansel
MBB Deutsche Aerospace

The Netherlands

Ir Maarten Boasson
Hollandse Signaallapparaten

United Kingdom

Mr Peter Chinn
Marconi Defence Systems

Prof. John T. Shepherd
GEC Marconi

Mr Kenneth A. Helps
Smith Industries/Aerospace in Defence Systems

United States

Mr Donald E. Dewey
Boeing Defense & Space Group

Dr Edwin B. Stear
The Boeing Company

Dr Anthony P. DeThomas
Wright Laboratory/Flight Control Division

Mr Stanley H. Wilson
Naval Research Laboratory/Computation Information Technology

Dedication

This Advisory Report of AGARD Guidance and Control Panel
Working Group 10 is dedicated to

Professor John T. Shepherd

An AGARD enthusiast, a dedicated long-term member of the AGARD Guidance and Control Panel, and a key member and contributor to Working Groups 6, 8, and 10. He was a natural technical leader with a keen intelligence, a visionary with a strong grasp of fundamentals, and a man of the highest professional integrity. Always a considerate man with a good sense of humour. A valued and trusted colleague whom the Panel will miss immensely.

Contents

Page

Page

Page

Page

Page

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Spectacular developments in electronics materials, devices, and circuits over the last 30 to 40 years have resulted in many vastly improved commercial and military systems. The new military systems included: avionics systems; navigation guidance and control systems; and command and control systems for aircraft, missiles, and spacecraft. A striking feature of the military systems was increased programmability, especially in those systems using digital computers and digital signal processors. Another striking feature was the movement, further upstream, of the analog-to-digital interface associated with sensors, so that more signal processing could be included in programmable digital signal processors. The increased programmability of these digital computers and signal processors, in turn, enabled development of new modes for different strategic and tactical situations in the military systems. Also, because of the increased programmability, new capabilities could be incorporated when further improvements in sensors, algorithms, and electronics hardware, as well as changes in system requirements occurred. However, as new capabilities were added, program size and complexity increased significantly.

Figure 1-1 illustrates the growth of onboard software for several U.S. Air Force aircraft systems developed over the past two decades. This growth has resulted from the greater sophistication of embedded systems and continued reliance on software to mechanize increasingly complex functions. This growth has caused the cost of developing these large systems to grow at an unexpected rate; according to the COCOMO cost model, the cost of producing a system with 1 million lines of code is 16 times greater than the expected 10 times the cost of producing 100,000 lines of code.

Our inability to estimate the software requirements for a given application is shown in figure 1-2; these data represent information gathered for seven actual systems. In each case, the schedule and corresponding costs increased in proportion to the software expansion. Additional data collected for these systems (fig. 1-3) show that 36% of the problems are attributable to translation of system requirements into software requirements and that 28% of the problems are attributable to design. The remainder of the problems are divided among several factors such as documentation, data, module coding, and interfaces. About 15% of the problems are corrected during software integration, 48% during system integration, and the remainder over the other phases of the development cycle.

In 1984, recognizing the need for more powerful design methods and support tools, the AGARD Guidance and Control Panel began investigating the state of software design and software engineering environments for navigation, guidance, and control systems. The results of that effort were summarized in the Working Group 08 report. The main conclusion was that, while current tools being developed will significantly improve the cost and integrity of software, this improved software will still not be able to meet future requirements. Clearly, additional and more powerful methods and tools are essential if software is not to become a major bottleneck to developing future military systems.

Prompted by analogous problems in business-related software, a new class of software design and implementation tools has been developed called software generators. These software generators are introduc-

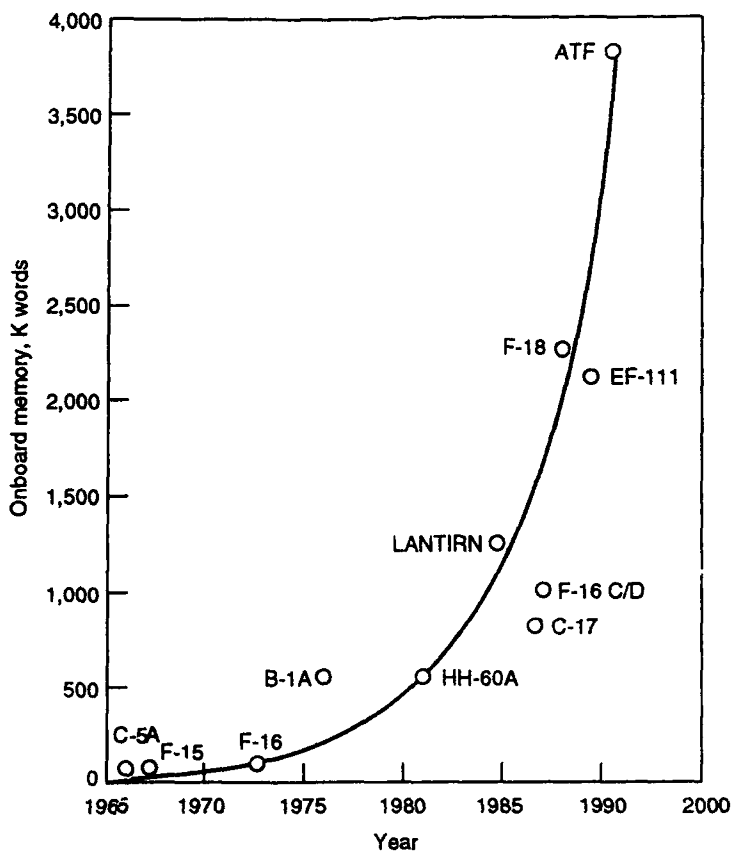


Figure 1-1. Weapon System Software Growth

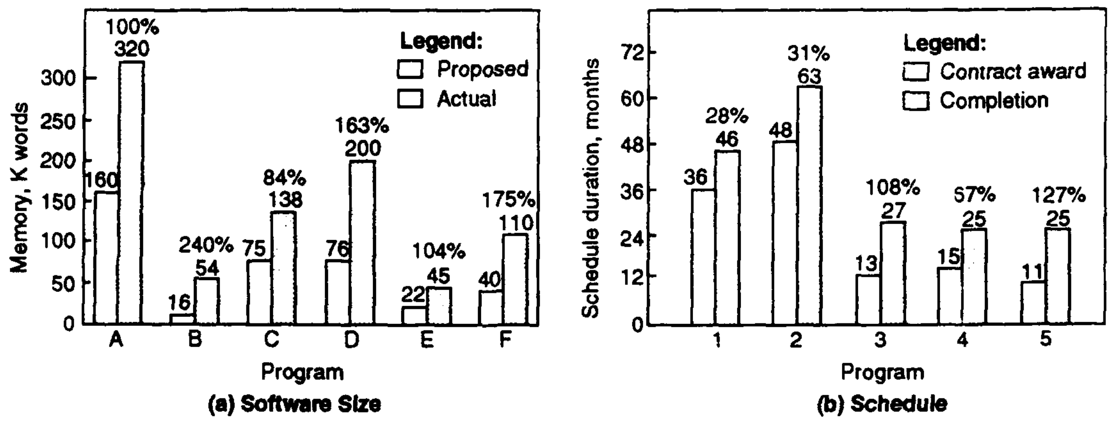


Figure 1-2. Software Size Growth

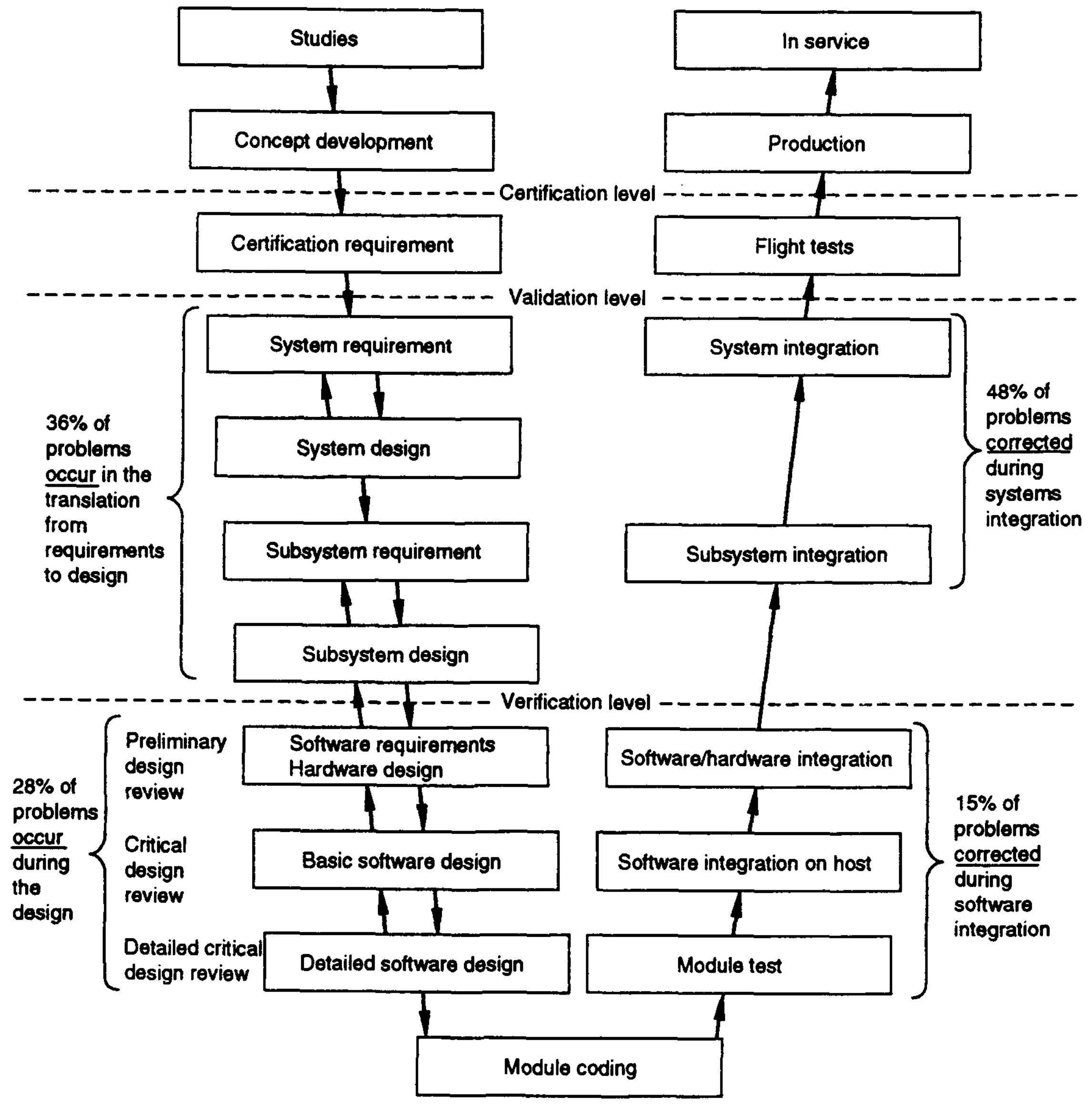

Studies
Concept development
Certification level
Certification requirement
Validation level
System requirement
System design
Subsystem requirement
Subsystem design
Verification level
Preliminary design review
Software requirements Hardware design
Critical design review
Basic software design
Detailed critical design review
Detailed software design
Module coding
Module test
Software integration on host
Software/hardware integration
Subsystem integration
System integration
Flight tests
Production
In service
36% of problems occur in the translation from requirements to design
28% of problems occur during the design
48% of problems corrected during systems integration
15% of problems corrected during software integration

Figure 1-3. Where Software Problems Occur and Are Corrected

ing much higher levels of automation in the software development process. Realizing that these generators are a normal evolution of software tools, the AGARD Guidance and Control Panel formed Working Group 10 to investigate the potential of software generators. This report contains Working Group 10's findings and conclusions.

This report has seven chapters. Chapter 1 is the introduction. Chapter 2 discusses the reusable software as the basis for software generation and chapter 3 discusses the expert systems approach to software generation. Chapter 4 discusses a program transformation approach to software generation and chapter 5 discusses the use of fourth generation languages as a basis for software generation. Chapter 6 discusses combinations of the four methods of software generation. Finally, chapter 7 is Working Group 10's conclusions from the study.

1.2 HISTORICAL BACKGROUND

Any proper discussion of new development techniques for navigation, guidance, and control systems software must take place within historical context of development of computer architectures and languages, software design and implementation tools, and software engineering environments.

1.2.1 Evolution of Computer Architectures

In the early development of navigation, guidance, and control systems, computations were implemented via analog devices and computers combined with simple networks to switch from one mode of operation to another. During World War II, and shortly thereafter, systems performed remarkably well using this limited and primitive technology. However, development of the stored program digital computer was bound to significantly impact future navigation, guidance, and control systems in many ways. Thus, it was not surprising to find these systems in some of the most advanced aircraft and missiles of the 1950s.

The first stored program digital computer was the Verdan computer developed by Autonetics. This computer contained digital differential analyzers which were digital implementations of analog integrators used in differential analyzers in World War II guidance and control equipment. In this same timeframe, Litton, Bendix, and other companies also developed digital differential analyzers for various pieces of equipment. However, these digital differential analyzers only replaced the integrator part of previous analog systems that solved navigation, guidance, and control equations. Thus, they were limited in their functionality and programmability. Hence, it was not long until digital differential analyzers were, in turn, replaced by stored program digital computers of the von Neumann architecture.

Early versions of these stored program computers were the IBM 4 Pi and the Delco magic computers which were widely used in digital navigation, guidance, and control systems implemented in the United States during the 1960s. This class of digital computers was used often in navigation systems and fire control systems, but almost never in flight control systems for manned aircraft. However, in the case of ballistic missile systems, such as the Minuteman, these digital computers were used throughout the total navigation, guidance, and control system implementation. The early Minuteman system was one of the first uses of stored program digital computers in flight control applications. Because of concerns with accuracy and proper functioning of nuclear tipped missiles, the integrity of ballistic missile guidance and control systems received a great deal of attention. (It is interesting to note that 40% of the cost of developing the software for these systems was devoted to independent verification and validation of the software. The general issue of software verification and validation was a Working Group 09 subject and further information is in that final report.)

The continued development of higher performance von Neumann computers in weapons systems has proceeded through the current generation. Most applications use only a few such computers which are connected and operated in a loosely coupled way. However, in flight control systems, these computers are connected in a redundant, tightly coupled configuration. Fault tolerant design through redundancy was pioneered by flight-critical flight control systems for aircraft. The high-integrity requirement sets such systems apart from other, less critical systems such as mission avionics and command and control systems. However, we are now witnessing a vast expansion of redundant, fault tolerant systems for applications other than flight control, such as nuclear reactor control and banking systems. Even more use of fault tolerance in both hardware and software can be expected in the future for applications, such as medical diagnosis systems and others, where high integrity is of the utmost importance.

Another trend, apparent in the generation currently being developed, is high-level integration using information from a wide variety of sensors and processors operating in parallel in either a closely coupled, synchronous mode or asynchronously. Obviously, developing high-integrity software for such multiprocessing systems operating in closely coupled or asynchronously redundant modes is an increasingly difficult problem. This problem must be solved if the full potential of modem sensor systems is to be exploited to improve military system performance. From a computer architecture point of view, redundant heterogeneous multiprocessors, operated either asynchronously or in tightly coupled modes, are the order of the day and form the context for future software development.

1.2.2 Evolution of Languages

Programmable system development brought the new task of efficiently programming these systems. In the earliest digital systems, programming was done directly in machine language through switches operated by a person at a console. That is, the 1s and 0s representing the program were entered directly into registers and, then, transferred into the computer's memory. This was a rather laborious job and it quickly became apparent that, if such computers were to have any real utility, a better means would have to be found to program them. Thus, assembly languages were developed in which abbreviated neumonics expressed the operations to be programmed into the computers. Then, these neumonics were translated into the machine language required by the computer.

Another early realization was that many of the operation sequences in programs were repetitive and that the programming task would be greatly aided by developing so-called macros. Macros are sequences of assembly language instructions which can be grouped together and invoked as units wherever they are needed in the program. Macros represented the first widespread use of reusable software. Assembly languages and their associated macros are very powerful compared to machine language and they enable great visibility into the actual operation of the digital computer. As a result, assembly language and macros are used in many high-integrity systems because visibility is critical to verify and validate the computer programs. Visibility is especially important in high-integrity, real-time systems.

From the beginning, most uses of digital computers involved scientific and business computations. As a result, it wasn't long until the first high-level languages appeared in Fortran for scientific calculations and Cobol for business applications. Fortran (formulation translation) was appropriate for transforming formulas for scientific computations into a high-level language. This language could subsequently be translated into an assembly or machine language by a piece of software or tool called a compiler. Fortran was followed shortly by Algol 60 in Europe. Various second generation languages were developed in the 1960s including PL1, Pascal, C, and many others. Other high-level languages which were developed in the context of military applications were JOVIAL Coral 66, Pearl, and LTR and its various derivatives.

More recently, third generation languages have been developed, such as Ada and LTR3. Developed specifically for real-time systems, these languages have several very desirable features including the ability to enforce desirable design methods.

All of the languages mentioned in this section are procedural languages in that they tell a computer how to do things instead of just what should be achieved. Coincident with development of these procedural languages was the development of nonprocedural languages which describe what is to be done, but not how to do things. These declarative or nonprocedural languages have an important role to play in the early design phases when specific algorithms for implementing the design remain to be determined. Development is expected to continue in nonprocedural languages associated with expressing requirements and specifications for software and hardware.

1.2.3 Evolution of Tools

Almost from the advent of programmable computers and systems, tools were developed to assist in the programming process. For example, for the Verdan computer, which had a drum base memory, a circular slide rule was developed which made it easy to determine which memory locations would be under the read/write head after the time required to decode instructions and fetch data. In essence, this was a tool to speed up the programming process and, also, to maximize the throughput rate. Similarly, development of assembly languages was accompanied by development of assemblers. This was followed by development of compilers (and optimizing compilers) for higher languages such as Fortran and Cobol to maximize computational speed and/or to minimize memory requirements. Other tools which became standard included editors, macrogenerators, linkers, loaders, etc. It is fair to say that any organization, seriously involved in development of software during the late 1960s and 1970s had at least a minimum set of these tools. However, in most cases, these tools were not integrated to any great extent. Also, in a great many cases, additional tools were developed by major companies developing software to support the efforts of that company's particular software groups. In the late 1970s, it was not uncommon to find groups with a tool set of 25 or more distinct tools. Clearly, development of tools to support software development has been underway essentially from the beginning. However, many of these tools originated with the specific programming groups and could not be easily adopted by other organizations or integrated with the existing tools that other organizations had.

In the 1970s, there was an increasing awareness that software was a big part of systems development id that much more powerful tools were needed to assist the programming process. It is clear today that, without a great deal more effort on tools that support and enforce particular design methods, we will not progress in software development to realize the potential from recent and continuing advances in hardware.

1.2.4 Evolution of Environments

There was a realization in the late 1970s that, to be really useful and efficient, the various software development support tools needed to be integrated. Integration was needed so that the ouput of one tool could be used as input by another tool and so that the right information would be captured by any tool in a form that could be used by other tools. The need for integration led to the concept of software engineering environments.

It was accepted that tools should be able to run under a common operating system or executive. Unfortunately, the various tools which had been developed ran under a variety of operating systems. Moreover, most operating systems did not have the most effective facilities for tool integration. The obvious solution to this problem was to build a programming support environment on top of existing operating systems which provided the necessary functionality to integrate existing and future tools. Therefore, the major part of this development of programming support environments required common widely accepted interfaces and was developed in a layered structure with a kernel system which interfaced most directly to the hardware system. The program support environment would run with higher level layers being built to interface with the kernel system through standardized interfaces enforced by the kernel. Of course, if this were to work in an integrated and portable way, there would be a need for development and wide acceptance of appropriate interface standards.

However, the importance of operating system and database structures that support software engineering environments was acknowledged. In the case of Ada, evolution of a standard support environment began with the Stoneman requirements. A great deal of effort went into developing software programming support environments based on the Stoneman specifications. The effort is continuing today. While a great deal of progress has been made, much more progress is required. It is probably fair to state that this task has proved to be more difficult than originally anticipated. The technology was oversold and it will take longer than originally planned to develop the kind of Ada programming support environments anticipated in the Stoneman specification. The significance of the database and operating system aspects of this problem should not be underestimated. A significant amount of attention paid to these aspects of the problem may pay off handsomely. In fact, it may even be required if a proper Ada program support environment or any effective program support environment is to be developed. Balzer made this point in his discussions of the next generation operating system with its associated intelligent database where demons do a great deal of work instead of work being controlled in a command sense. The kinds of techniques discussed by Balzer will be detailed as part of the program transformation approach to software generation.

1.3 TYPES OF GUIDANCE AND CONTROL SOFTWARE

In Working Group 10's consideration of the role of software generators for future navigation, guidance, and control software development, it was concluded that it would be useful to describe the types of navigation, guidance, and control software which are required to carry out the design, implementation, and postdevelopment support of software for this particular application area. A list of the various types of software required for the design, implementation, and postdevelopment support of navigation, guidance, and control systems is in table 1-1.

1.3.1 System Design Software

System design software can be described as that software which is used by the navigation, guidance, and control engineers directly in the design of systems and software at the highest levels. Four specific classes of tools can be identified to support the system design process. These include requirements development tools, concept formulation and development tools, design tools, and design capture and documentation tools.

The requirements given to the navigation, guidance, and control engineer at the earliest design stages are very general. The task is to conceptualize and carefully describe a set of alternative system concepts that could satisfy the overall requirements. Also, the task is to analyze the various alternatives and decide which best meets the requirements and should be detailed and passed on to the next stage of the navigation, guidance, and control system design. At this stage, there is no sharp distinction between the realization of navigation,

Table 1-1. Types of Guidance and Control Software

• System design software • Requirements development tools • Concept formulation and development tools • Design tools • Design capture and design documentation software	• Controls and display software
	• Mission planning software
	• Software design and implementation software and management support software
• System simulation software • Vehicle simulation software • Navigation, guidance, and fire control system simulation software • Pilot-vehicle interface simulation software • Mission execution simulation software	• Guidance and control system verification and validation software
	• Guidance and control system certification software
	• Automatic test equipment software
• Operational flight program software • Flight-critical software • Mission-critical software • Run-time environment software • Operating system/executive	

guidance, and control laws in hardware or software. Most of the effort is directed toward defining the sensor complex required to support sensor and actualization complexes to support the high-level design and the top-level functional and data flow descriptions to implement the overall system.

The system design software is like design support software; it consists of tools that do not necessarily run in real time and that can support the kinds of analysis and synthesis for a final design and implementation. The four kinds of software tools appropriate for this system design and implementation process are discussed in the following four sections.

1.3.1.1 Requirements Development Tools

Tools to support requirements development are essentially nonexistent. All known software tools of this kind were summarized in the final report of Working Group 08.

Independent of the software development tool business, which is spearheaded by the computer science community, the navigation, guidance, and control systems community has been developing tools and environments to support the top-level design in guidance and control systems. Prime examples of this are the computer-assisted control system design (CACSD) developments, several of which are available on the market. There is currently great activity in this area, with rapid improvements in existing CACSD environments and development of new ones. Obviously, these developments need to be brought in at the system design level. However, it should be kept in mind that the people using these tools are navigation, guidance, and control engineers and not necessarily software specialists or programmers. Therefore, whatever tools and environments are developed need to support the syntax and semantics commonly used in CACSD systems.

1.3.1.2 Concept Formulation and Development Tools

Concept formulation and development tools are also a new area with the CACSD developments being an initial attempt at providing such tools. The Working Group 08 report describes other tools, if any, that may be available for this stage of the process. Beyond general observations, the kinds of tools desired at this level are those that would support entity relationship, object-oriented design, and functionally oriented design concepts. Much of the concept formulation at this stage has traditionally been done in block diagrams and other graphics-oriented tools. It is expected that such tools in the future will also have a strong graphics component. Further, in chapter 5 of the Working Group 08 report, it is stated that graphic representations should also be supported by machine processable textual equivalents. Other useful ways of describing systems at this general level may be developed in the future. If so, such concepts should be incorporated in concept formulation and development tools that are used at the overall system design level.

1.3.1.3 Design Tools

Design tools can be broken into two major classes; analysis tools and synthesis tools. Navigation, guidance, and control system design, at the level being discussed here, has been done largely by iterative design and analysis of system concepts that were based on historical experience and research. A wide variety of analysis tools are available including: block diagram manipulation; various stability and response analysis tools (linear systems analysis tools, such as frequency response methods and Laplace transform methods including root loci); and various other stability and time-response analysis methods. In addition, there are direct time-response techniques based on direct use of vector-matrix descriptions of dynamic equations. Also, some of these analysis techniques can be used to analyze nonlinearities in the systems hypothesized. Essentially, what is being described here is the standard array of analysis tools that has been developed by the navigation, guidance, and control engineers over the years as well as the customization of these techniques to exploit modern powerful digital computers.

A major aspect of analysis tools (which will be deferred until later) involves simulation. Simulations represent specific software developments. Many of the analysis tools discussed here are in the CACSD systems referred to previously. In fact, the CACSD focus has been to develop suitable environments for invoking analysis tools in an integrated fashion.

In addition to analysis tools, synthesis tools should also be considered although this area is not as promising. While there have been attempts to develop powerful synthesis tools, little real success has been achieved. As a result, this need still exists although some limited progress has been made (such as synthesizing filters for feed-forward and feedback loops and design of Kalsman filters for certain applications). Generally speaking, direct synthesis tools at this level would be very powerful design tools.

1.3.1.4 Design Capture and Design Documentation Software

Obviously, design capture and documentation software at all stages of design and implementation is needed. Design capture and documentation software means tools which can capture design concepts for analysis and synthesis and produce a design description that can be used for later implementation and postdevelopment support. It is not enough just to capture the design. It is also important to document the design, so that the reasons for design decisions will travel with the design itself. Then, the design rationale will be available for the inevitable postmodifications which occur during design-development support of navigation, guidance, and control system software. Design capture has its counterpart in electronic circuit design that relates to graphic design descriptions favored by navigation, guidance, and control engineers. Clearly, design capture and documentation occurs at all design and implementation levels including the software design process.

Hopefully, the efforts made in design documentation software for electronic circuit design can be adapted to this level of system design as well.

From a database viewpoint, the design should be captured so that it can be elaborated in the future steps of detailed design and so that all aspects are available for later stages of design and implementation. Moreover, it is important that the overall system be captured as a single data representation from which all other required representations can be derived. This leads to important considerations regarding the database concept for design and implementation. These particular database issues are not unique to navigation, guidance, and control systems, but they need some customizing. For example, it is necessary to incorporate those features which relate to the time-critical and flight-critical nature of the systems and to properly represent and cope with the redundancy and fault tolerance aspects as well.

1.3.2 System Simulation Software

Throughout the system design and implementation process, simulation software is needed. At the top level of system design, software is needed for: vehicle simulation; navigation, guidance, and fire control algorithm simulation; pilot-vehicle interface simulation; and mission execution simulation. Moreover, as the design and integration process proceeds, the amount of purely digital simulation rather than high-fidelity, hot-bench, and ironbird simulation, decreases. It is important in the development and support of simulation software that such changes be easy to implement and automate.

1.3.2.1 Vehicle Simulation Software

At the highest level of design, simulation is done within the digital computer. For example, in the case of vehicle simulation, software can be identified for: kinematics and dynamics of the vehicles; representation and simulation of the aerodynamic and propulsion characteristics of the vehicles; simulation of the vehicle sensor systems; and simulation of stability and control characteristics associated with the vehicles. Simulation software is typically implemented in higher order languages. Primarily, simulation is used to evaluate the significant complex interactions and nonlinearities of the system including limits on the vehicle. In a sense, this software directly supports the evaluation of various navigation, guidance, and controls concepts. Vehicle simulation programs can include a great deal of data and these data will change during the evolution of the overall system. Therefore, changing, verifying, and validating data must be convenient and easy. This involves forming an interface with those individuals associated with the design of the vehicle. It is also necessary for the algorithms involved in such a simulation, particularly the integration algorithms, to be able to handle a wide variety of time constants (stiff systems) if they are to be effective. As much as possible, the decision about which algorithms to use should be automated or determined by the simulation software itself.

1.3.2.2 Navigation, Guidance, and Fire Control System Simulation Software

In addition to vehicle simulation, it is necessary to efficiently simulate navigation, guidance, and fire control laws developed during system design. These simulations should be easy to modify during design. For simulation of the outer-loop functions associated with navigation, guidance, and control systems, it should be possible to automatically extract the parameters directly from the design capture process. And it should be possible to automate transformation from this level of simulation directly to implementation of the operational flight program software.

These simulations are used to: verify the performance of the navigation, guidance, and fire control system concepts derived at the top level of system design; tune the parameters of navigation, guidance, and fire control laws to achieve acceptable performance; and determine performance limitations inherent in the navigation, guidance, and fire control algorithms (laws). The syntax and semantics of this level of simulation should be as close as possible to that used by navigation, guidance, and control engineers. Ideally, it should be possible to automate the operational flight program software development based directly on the simulation at this level.

1.3.2.3 Pilot-Vehicle Interface Simulation Software

In the case of manned systems, an additional major piece of simulation software is required for the pilot-vehicle interface. This simulation includes two kinds of software; control and display simulation and pilot-in-the-loop simulation software. Obviously, such simulations should be generic and customizable to a wide variety of control and display systems. These simulations present the opportunity to do prototyping in its finest sense; that is, to define the pilot-vehicle interface that will meet the overall system requirements. In a properly designed simulation environment, it should be possible to go from control and display simulation parameters directly to the generation of the control and display software required in an operational system.

Pilot-in-the-loop simulation determines the pilot's ability to cope with the system concepts that are being proposed. Hence, this software must be able to exploit the pilot's capabilities in all aspects of the navigation, guidance, and control of the vehicles involved. Clearly, this simulation software has to interface with the navigation, guidance, and fire control system simulation to evaluate the pilot's overall performance and mission effectiveness.

1.3.2.4 Mission Execution Simulation Software

In addition to the above kinds of simulation software, mission execution simulation software is required to simulate all other aspects of mission execution important to the system. The nature of this software varies dramatically for missiles, aircraft, and spacecraft. In any case, this software must be integrated closely with other simulation software to achieve an overall mission simulator capability to evaluate the system performance at various levels. The functions of such software could include, for example, various communications and information from outside the system to permit coordinated attacks between separate vehicles as well as various kinds of software to simulate queuing from command and control systems, etc.

1.3.3 Operational Flight Program Software

There are four kinds of operational flight program software; flight-critical software, mission-critical software, run-time environment software, and operating system/executive software. Typically, each of these kinds of operational flight program software is developed by independent groups based on individual specifications. Each of them is a candidate for software generation.

1.3.3.1 Flight-Critical Software

Flight-critical software consists of the traditional flight control software, together with software for flight control sensor signal processing and for flight-critical functions, such as terrain following and terrain avoidance. This software has the highest integrity requirements and any software generators used to generate such software would have to be of equal or greater integrity.

1.3.3.2 Mission-Critical Software

Mission-critical software consists of all software that is essential to maintain total mission function, but that is not flight critical. Typically this consists of navigation and guidance software, fire control software, and most avionics system software. Mission-critical software is typically much larger than flight-critical software and has much less of a requirement for integrity.

1.3.3.3 Run-Time Environment Software

Run-time environment software includes all of the software associated with sensor and actuator input and output (I/O). This can be quite extensive in navigation, guidance, and control systems that interface with a wide variety of sensors and actuators, as well as pilot controls and displays. Like flight-critical software, run-time environment software typically has a high-integrity requirement as well as stringent requirements on execution times.

1.3.3.4 Operating System/Executive

Operating system/executive software is particularly critical, especially if it is associated with flight-critical software and fault tolerant, redundant systems. Moreover, this software is becoming increasingly complex in the heterogeneous software environment now found in most military systems.

1.3.4 Controls and Display Software

Controls and display software consists of software associated with cockpit controls and displays. It consists of symbol and display generation software, control input processing software, and bus control software. These three kinds of software are potential candidates for software generators.

Control input processing software is relatively straightforward to implement, and should be the easiest kind of software to generate automatically. Symbol generation software is somewhat more complex, but should also lend itself well to software generation, especially in the context of graphics and symbol generation standards. Bus control software, including bus protocols, is already being investigated as an area for software generation.

1.3.5 Mission Planning Software

Mission planning software output typically consists of flight routes or flightpath parameters which are entered into onboard systems involved in mission execution. In the case of navigation, output would consist of waypoints, altitudes, program turns, etc. In the case of fire control systems, output would consist of various parameters associated with target attack as a function of the practical situation and the weapons used. In the case of missiles, output might consist of appropriate guidance and control gain schedules, maneuver loads as a function of weapon loads, etc.

1.3.6 Software Design and Implementation Software and Management Support Software

Software design and implementation software is a set of software tools used for navigation, guidance, and control system software design and implementation and any software engineering environment-related software. This software is used in the initial design and implementation and in the postdevelopment support phase.

Management support software is those software tools used to support the management of software design and implementation. It includes cost estimating tools, configuration control tools, scheduling tools, etc.

1.3.7 Guidance and Control System Verification and Validation Software

Guidance and control system verification and validation software ranges from specification tools through test generation tools. In general, each one of these tools is a candidate for software generation.

1.3.8 Guidance and Control System Certification Software

Guidance and control system certification software supports the certification process. These tools are mostly for requirements and specification tracing, special documentation, etc.

1.3.9 Automatic Test Equipment Software

Automatic test equipment software is used to program automatic test equipment, to execute proper test sequences, and to record test results. It has typically been programmed in the standard high-level language ATLAS. However, recent standardization thrusts are promoting the use of Ada.

1.4 CONTEXT

To evaluate the potential for software generation of navigation, guidance, and control software, it is important to have some context for this particular application. This context is described in table 1-2.

Table 1-2. Context for Real-Time Software Generator Development

• Imbedded software	• Hard real-time constraints
• Operational life	• Execution speed and memory constraints
• Updates	• Integrity and fault tolerance
• Configuration management and control	• Role of systems and applications programmers

1.4.1 Embedded Software

Navigation, guidance, and control software is embedded in the sense that it represents an essential, but not dominant part of the whole system. Thus, any consideration of software generators must be in the context of the overall system design, including the software, with the full realization that hardware parameters, system performance parameters, algorithms, and data are essential input to the generators themselves.

1.4.2 Operational Life

The operational life of typical navigation, guidance, and control systems is 10 years to a few decades. Thus, it is essential that software generators support this software over long operational lives.

1.4.3 Updates

Navigation, guidance, and control software is characterized by frequent updates to correct errors discovered during the operational phase, as well as to incorporate new capabilities as a result of new requirements and/or improved hardware capabilities. Over the life of the system, there may be a great many updates and the overall postdevelopment activity may well exceed the original activity.

1.4.4 Configuration Management and Control

Configuration management and control is particularly important in the development of navigation, guidance, and control software. This is because the software is complex, it is developed by many different teams of individuals, and its integrity is paramount. Thus, use of software generators must lend itself to strong configuration management and control discipline.

1.4.5 Hard Real-Time Constraints

The execution of guidance, navigation, and control software must satisfy hard real-time constraints, in order to satisfy response time and stability requirements. Therefore, software generators for these systems must be able to accept hard real-time constraints as input and must be able to determine execution times on the target processor when the software is being generated. This requires that a good timing model of operation execution on the processor or multiprocessors must be available to the software generators.

1.4.6 Execution Speed and Memory Constraints

Execution speed of the processor used in navigation, guidance, and control systems continues to increase over time and the cost of memory continues to decrease over time. However, execution speed and memory constraints continue to be important because of the increase in system functions, and their complexity. This implies that software generators for navigation, guidance, and control systems software will need to produce both time- and space-efficient code.

1.4.7 Integrity and Fault Tolerance

Navigation, guidance, and control systems require high integrity. In addition to fault avoidance, the integrity requirement most often leads to redundancy for fault tolerance. Redundant fault tolerant architectures can be exceedingly complex and demanding of the design and implementation process. To be widely accepted and truly effective, software generators for navigation, guidance, and control systems must be able to produce software for redundant, fault tolerant, heterogeneous multiprocessors.

1.4.8 Role of Systems and Applications Programmers

Software generators have the desirable effect of bringing the software implementation process closer to the systems designers. Ideally, the software generator would allow the systems designer to implement the software directly from a correct specification from the systems level design process. Clearly, if software generators are to fulfill this role, they must use syntax and, especially, semantics that are appropriate to a particular class of systems and applications. In this division of labor, computer scientists should concentrate on developing the technology for the design and implementation of software generators (using an appropriate syntax and semantics suited to the applications area) and leave the actual software specification and implementation process to the systems engineers who understand the required functionality and performance. In this context, software generation is done by one more tool at the disposal of the systems engineer.

1.5 ISSUES

There are many issues to consider in developing software generators. Although these issues vary somewhat for particular kinds of navigation, guidance, and control systems, many of them are the same. A somewhat extensive list is in table 1-3. A detailed discussion of these issues is in appendix A.

Table 1-3. Issues in Real-Time Software Generator Development

• Design methods	• Supportability/ease of modification of both applications and support software
• Requirements capture	• Support environment requirements
• Role and use of prototyping	• Documentation
• Task assignment in redundant fault tolerant multiprocessor architectures	• Configuration management and control
• Necessity and structure of alternate viewpoints	• Generator integrity
• Functional versus object-oriented design	• Asynchronous versus synchronous systems
• Assisted versus automatic generation	• Concurrency
• Internal communication mechanisms	• Run-time environment: generator definition, design, and description
• Role and use of intermediate languages	• External communication mechanisms
• Procedural versus nonprocedural languages	• Dynamic memory allocation
• Syntax handling	• Proprietary limitations
• Semantics handling	• Database attributes

1.6 APPROACHES TO SOFTWARE GENERATION

Working Group 10 has defined four approaches to software generation: reusable software modules, expert systems, program transformation techniques, and fourth generation languages.

1.6.1 Reusable Software Modules

The reusable software module approach to software generation uses a set of standard software modules which are automatically combined into an application software program by a composition process. This approach is analogous to silicon compilers using standard cells which are combined to form integrated circuits. In this analogy, the standard cells correspond to reusable software modules. This approach requires a powerful and complete set of modules which are completely characterized in terms of function, execution, and performance with well-defined input and output interfaces. In addition to the reusable software modules, the key additional piece of software is the module composer. The module composer has to be able to analyze the software specification; determine which modules are required to optimally, or at least adequately, implement it; and connect the modules with suitable control and data flow to realize the required specification. The modular composer may also have to include analysis and simulation capability to verify performance of the resulting software program.

1.6.2 Expert Systems

In its more primitive form, the expert system approach provides expert assistance instead of automatic software generation. In its more mature form, this approach would totally automate the software generation process. Per current expert systems technology, this approach would be implemented as an expert rule-based system in any one of a number of high-level languages. Effort has been underway for some time to develop a knowledge-based software assistant. What exists now may be the best that can be realized and complete automation of the software generation process may never be realized using this approach.

1.6.3 Program Transformation

The program transformation approach to software generation is based on formal techniques, including a formal specification, followed by a set of successive program transformations which invokes proof-of-correctness principles at each transformation. While this approach may ultimately prove to be the most flexible and highest integrity approach, it is currently limited by lack of adequate technology to support formal specifications and proof of correctness. Current technology can only support the smallest programs. Fortunately, the highest integrity guidance, navigation, and control software (that is, flight-critical software) is typically small. As a result, program transformation techniques may find their first success in high-integrity, flight-critical systems.

1.6.4 Fourth Generation Languages

Fourth generation languages are currently very successful in business applications. Essentially, fourth generation languages can be characterized as application specific languages which capture the syntax and semantics for the particular application area for which they are developed. As a result, fourth generation languages are usable directly by the applications experts without the help of systems and applications programmers. It is this responsiveness and the elimination of the intermediate level of programmers that make fourth generation languages so popular. There is no fundamental reason why powerful fourth generation languages cannot be developed for navigation, guidance, and control systems. In fact, some initial efforts to develop such languages are underway. It is clear from common sense reasoning and the experience of people working in this area, that fourth generation languages for navigation, guidance, and control systems will have to incorporate the syntax and semantics commonly used by navigation, guidance, and control engineers. Because these engineers use specialized graphics techniques, it is also clear that powerful graphics techniques and languages will have to be used in this approach to software generation. For technical reasons, it is important to have equivalent text representations for all graphic representations used.

CHAPTER 2

REUSABLE SOFTWARE APPROACH TO SOFTWARE GENERATION

2.1 OVERVIEW

An effective way to reduce the cost of complex systems is to *buy* rather than *build* as many software components as possible. *Buying* in this sense means reusing existing software designs and components.

Reusability means using a software product in multiple applications. The concept of reuse in nonsoftware engineering disciplines, such as electronics, is well understood; for example, radios can be built for one application, but used in many other applications. In addition to reusing an entire product, reuse can refer to reusing standard electronic designs, methods, or, components to build new products. Standard, off-the-shelf transistors, microprocessors, and mounting racks are examples of reusable components; and amplifier design and construction procedures are examples of reusable methods.

Obviously, it is impractical to design and develop new components and methods if they already exist and the designer knows where to find them. Often, in software development, the designer does not know where to find the components or does not have adequate information to make a decision about whether to use the components.

To fully exploit software reuse, the reuse system must have certain characteristics. Among these are mechanisms for (1) specifying a reusable object, (2) finding the object, (3) modifying the object, and (4) knowing how to use the object to compose new systems. Finding a reusable object implies using a software library. This library must provide a specification that permits an unambiguous understanding of the object by a broad experience base set of users. Facilities must also be provided for tailoring the object to meet the requirements of a particular application domain. Finally, tools must be established to compose and test a system formed from the reusable object.

Like hardware, reusable software can include objects from all phases of a development cycle: requirements specifications, software design (including system modularization and module specification and design); PDL, object, or source code format; development tools and test tools; as well as actual PDL, object, or source code. Reusability can be applied in all major stages of the software life cycle to improve productivity, quality, portability, and development cycle time. Reusability, in fact, is already being applied in many instances.

Current reusable software techniques can be divided into three categories: (1) efforts that concentrate on designing software and systems so that the software modules can be reused, (2) software module cataloging and library systems (including efforts that generate reusable modules for the express purpose of populating the library), and (3) systems for examining existing software to select candidates for reusability.

Another method of classifying reuse is by the type of program reused. Reuse of mathematics and utility functions is common. All systems have standard subroutines for this reuse. Because there have been standard interfaces to such models for some time and because the functions and possible variations are well understood, their use is easy to specify. The next most obvious set of functions to standardize and reuse are those relating to databases and to manipulating data in structured files. For example, Generic, Reusable Ada Components for Engineering (GRACE)[1] provides standard modules to manipulate various data structures in an Ada language system. A further categorization of reuse methods is according to dependency: that is, whether the method is

[1] *A description of the systems cited in this section is in section 2.2.*

language dependent (for example, Common Ada Missile Program (CAMP) and GRACE are Ada-dependent); methodology dependent (Software Cost Reduction Methodology (SCR); tool dependent (CARE); or application dependent (CAMP). Obviously, such dependencies are important in choosing a reuse doctrine or methodology.

In examining modules (or designing modules for reuse), it helps to have a set of characteristics and metrics to judge a module's suitability for reuse. Such a set was derived by the Software Technology for Adaptable Reliable Systems (STARS) program and is described in section 2.2.1.2.

2.1.1 Reusability Issues

Although the concept of software reuse has existed for some time, NATO countries have only recently begun to exploit it. The reasons for not exploiting this concept sooner include the following:

a. The *not-invented-here* syndrome: designers may not trust software that was written by other people.
b. The mistaken idea that top-down design precludes the identification and use of lower level software units.
c. The actual or perceived need for efficiency, leading to custom modules written in low-level machine-dependent languages.
d. The software specifications are either nonexistent or ambiguous enough that it is not possible to determine exactly what the software does without examining all of the source code. However, if the software is complex enough to make reusability attractive, it is complex enough to thwart any effort to infer its behavior in this fashion.[2]
e. The software performs a specialized task that resembles the required task, but the cost of customizing the existing software is greater than the cost of writing new software.
f. Although the software to perform the required task may exist, nobody on the new project knows about it or those who know of its existence don't know how to find it.[3]
g. Software that can perform the required task is available, but it is so general that it is inefficient for the task.

The major reason that most of the normal software produced today for guidance and control systems cannot be reused is because reuse was not a consideration when the software was produced. For software to be reusable, certain design and programming disciplines must be followed while the software is being produced. This is not a great burden on the software producers, however, because the same good design principles that allow the software to be designed, built, and tested rigorously are precisely the ones that allow the software to be reused easily. Software development methodologies that enhance software maintainability are particularly useful in building and cataloging software modules for reuse.

2.1.2 Potential Payoffs

Experience with reusable software for guidance and control systems is very limited; however, some fundamental activities are progressing. Existing data, although limited, provide projections for potential payoffs. As an example, table 2-1 illustrates the current development cycle for guidance and control systems along with the development weight assigned to each phase of the cycle. The development weight represents the percent of the total development effort related to a particular phase. The table also shows the range of expected improvements that might be achieved by application of reusability and a range of resultant savings

[2] *There is an effort (described in section 2.2.2.3) at the University of Maryland with the goal of computer-aided inference of the meaning of existing code by the examination of the source code.*
[3] *There are current efforts to establish libraries. See section 2.2.1.3.*

Table 2-1. Software Development Productivity Improvement Estimates

Development phase	Cost (%)	Improvement with reusability (%)	Resulting cost (%)
Concept definition	5	0	5
System design	12	20	10
Software requirements	14	40	9
Software design	15	20-60	12-6
Coding	10	50-80	5-2
Software unit test	12	20-80	10-3
Integration test	7	40	5
System test	10	30	7
Documentation	15	20-50	12-8
Total	100	25-45	75-55

over the development phase for a typical aircraft system. Table 2-2 illustrates similar information for the total software life cycle. The tables are derived from informal discussions with several guidance and control system software engineers in the industry and reflect a range of conservative to optimistic estimates derived from limited application experience and studies. These estimates are based on software not developed for reuse and not using object-oriented design. Based on the data, the development cycle costs could be improved by 25% to 45%, and total life cycle costs would show an improvement of 11% to 26%. Reusability in the development phase could account for a 5% to 9% improvement in life cycle costs.

The savings implied in these data can only be achieved after an investment is made to establish a library of reusable software elements including verified designs and algorithms, tested code, and methods for easy

Table 2-2. Total Software Life Cycle Productivity Improvement Estimates

Life cycle phase	Cost (%)	Improvement with reusability (%)	Resulting cost (%)
System development	20	25-45	15-11
Installation	5	0	5
Post-software deployment: • Defect removal	2	0	2
• Enhancements	58	5-20	55-47
• Functional redesign	15	20-40	12-9
Total	100	11-26	89-74

identification and retrieval of required elements. Other requirements include adequate training of personnel, discipline in using software methodologies, and management visibility into the process. With a well-established library of software objects and a means for easily obtaining knowledge of the objects, reuse can influence most of the development cycle phases. Typically, within a particular application domain, a common structure can be tailored to develop the new application. At the software requirements and design stages, which are typically done in a top-down manner, knowledge of available software elements at the lower level can aid in convergence to a solution. Although coding represents a relatively small portion of the development cycle, the reuse of tested code can affect not only the code but also the unit testing phase. Integration and system testing will benefit from the reuse of verified code because testing efforts can focus on system-level activities and not be plagued with lower-level problems.

A model for examining the relative software development cost of applying reusable software is given by:

$$C = (1 - R_N - R) + (R_N E/N) + R(b + E/n)$$

where C = development cost relative to a completely new product.

R = proportion of reused software ($R \leq 1$).

R_N = proportion of new element developed to be reused on other projects ($R_N \leq 1$).

E = relative cost of producing reusable software ($E \geq 0$).

N = number of uses over which R_N is to be amortized.

n = number of uses over which R is to be amortized.

b = cost of incorporating reusable software into product ($b < 1$).

The first term in the equation represents the cost of the nonreusable element, the second term is the cost of developing a new reusable element, and the last term is the cost of using previously developed elements. The corresponding relative productivity can be expressed as $P = 1/C$, where P = productivity relative to a completely new product.

The model relates development productivity to the proportion of reused software and associated development cost and cost of incorporation into a new software system. The model can be an effective tool for trade studies and sensitivity analyses to examine economic benefits and to determine the parameters having the greatest effect on the relative costs of reusing software objects.

Several cases can be examined to study the effects of the variables on productivity:

Case A: $R_N = 0, n >> E$

In this situation, $C = 1 - R(1 - b)$. As the relative cost of incorporating reusable software becomes high, the overall savings become small no matter what proportion of reusable software is used. Likewise, if the cost to reuse is relatively low, the cost savings are directly proportional to the amount of reusable software applied to the project.

Case B: $R_N + R = 1, N = n$.

In this ideal case, all the software developed is reusable, so that $C = E/n + Rb$. The cost is then the amortized cost of the newly developed reusable software and the cost of reusing previously developed elements.

Case C: $R_N + R < 1, N = n$.

For this typical case, $C = R(b + E/n - 1) + R_N(E/n - 1) + 1$. By setting $C = 1$, the payoff threshold (N_o) for reuse can be established as

$$N_o = \frac{E}{1 - b\left(\frac{R}{R + R_N}\right)}$$

Similar payoff thresholds for the other special case can be established.

To use the model, some initial estimates of various parameters must be made. For example, the magnitude of b is dependent on the level of abstraction of the reused element (e.g., requirements, design, code). To illustrate this process, consider the simplified software development cycle example in table 2-3. This development cycle consists of requirements, design, implementation, and testing phases.

The relative cost corresponds to the weight value in the previous tables. If unmodified actual code is used, then the only parts of the development cycle expected to apply would be the testing phase (T), with a value of 0.3. On the other hand, if the reusable element is a requirements definition only, then the design, implementation, and testing would have to be done, giving b a value of 0.8. For a new software element designed to be reusable, one would expect the relative cost (E) to be greater than 1 since it would go through all the normal development cycle phases as well as additional steps, such as insertion into a library.

Table 2-3. Software Development Cycle and Cost Example

Activity	Relative cost
Requirements (R)	0.2
Design (D)	0.3
Implementation (C)	0.2
Testing (T)	0.3

To illustrate a specific case for the model, assume b = 0.6, E = 1.5, $R_N = 0.1$, and N = 10. For these conditions the relative cost function becomes:

$$C=0.915-R\left(0.4-\frac{1.5}{n}\right)$$

and is shown in figure 2-1. As shown, if there is low reuse, the relative cost of development will be higher than not using reusable software because of the larger investment in developing the reusable elements. However, once the payoff threshold is reached, the cost will decrease with incurred use of reusable software.

The payoff threshold relationship for a sample set of conditions is shown in figure 2-2. As expected, as the cost to develop reuseable software and the cost to incorporate that software into a new system increases, the number of reuses required to recover development costs increases.

The sensitivity of cost to several of the cost model parameters was calculated for a specific set of conditions. The sensitivity is defined as the ratio of the per unit change in cost to the per unit change in one of

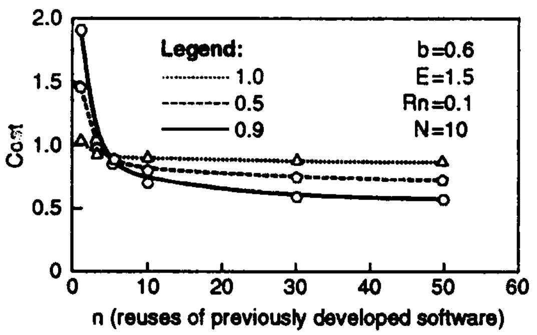


Figure 2-1. Cost Model Results Example

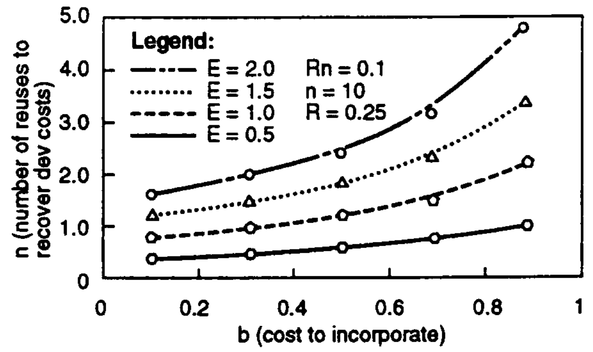


Figure 2-2. Payoff Threshold

the cost model parameters (for example, $(\Delta c/c)/(\Delta n/n)$). Figures 2-3 through 2-6 show that this type of analysis can be used to perform cost trade studies.

The sensitivity of cost to software reuse is shown in figure 2-3. The curves indicate that cost is most sensitive during early reuse. The breakpoint for the cost sensitivity function depends on the amount of software reuse.

The comparison of cost sensitivities to the cost of incorporation of reusable software is in figure 2-4 and the cost to develop reusable software is shown in figure 2-5. The data shows that cost will increase proportionately to the development and incorporation costs and to increases in amount of reusable software used.

The sensitivity of the cost to newly developed software is shown in figure 2-6. The pattern closely follows the pattern for previously developed reusable software shown in figure 2-3.

2.2 CURRENT STATUS

There are a number of efforts that concentrate on designing systems initially so that the modules can be reused. Examples include CAMP, Ada, some of the STARS efforts, and the SCR project.

CAMP identifies the modules that are common between missile systems and then develops modules that are usable in all or most of those systems. Of course, proper software modularization is critical to the program's success and is supported by the Ada packaging concept.

The STARS program is a U.S. Department of Defense (DoD) research program aimed at dramatically improving in software quality and development productivity and, at the same time, reducing the life cycle costs of embedded and mission-critical software systems.

The essence of the SCR project is the proper modularization of all phases of the software development so that changes to the requirements and design can be made easily. This *design for change* philosophy leads to modularity and specificity that enhance the probability of reusing the code.

Efforts that concentrate on cataloging the modules include CARE and GRACE. The purpose of CARE is to increase reuse of existing software components. The process defines and then measures characteristics that make components more reusable. The selected components are then made more reusable by writing a set of specifications. CARE has quantified the notions of small program size, simple structure, good documentation, and suitable language into a set of parameters that are used to

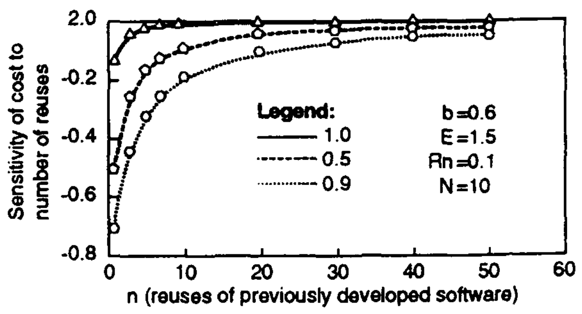


Figure 2-3. Sensitivity Analysis—Cost Versus Number of Reuses

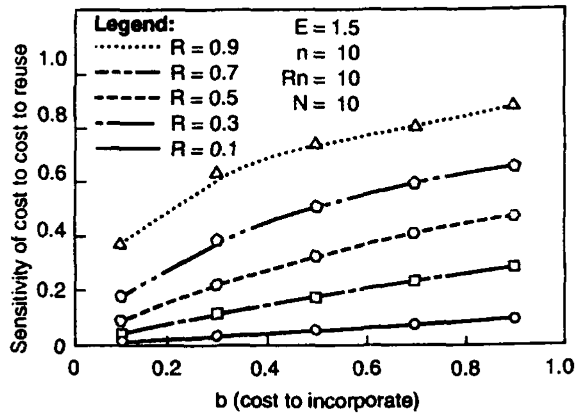


Figure 2-4. Sensitivity Analysis—Cost Versus Cost To Incorporate

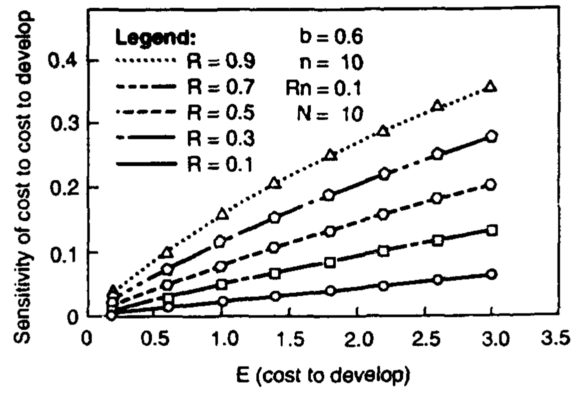


Figure 2-5. Sensitivity Analysis—Cost Versus Cost To Develop

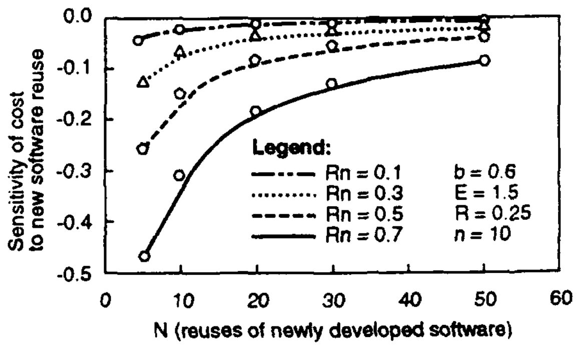


Figure 2-6. Sensitivity Analysis—Cost Versus New Software Reuses

measure components within the software factory. These parameters are used to identify reuse components. GRACE provides a set of Ada modules that manipulate particular data structures.

Other than the routine use of mathematical subroutines and utilities, the methods for reuse are either (1) considering reuse during original software design such as in CAMP, SCR, GRACE, and SPC or (2) concentrating on methods of capturing existing software regardless of its original design goals, inferring its specifications and appropriateness, and then cataloging the module for future reuse, such as in CARE. It seems evident that the greatest chance for achieving significant success with software reuse involves those methods in which reuse is a primary design goal in the original software. For software to be reused efficiently, the specifications must be precise enough to determine the intended results of executing the module and any possible side effects. Specifications of this sort are generally not available for systems that were not designed for reuse. As pointed out earlier, if a software module's function is significant enought to warrant reuse, then it is likely to be too complex for its total behavior to be inferred. Hence, it seems unlikely that any scheme to examine existing modules to pick candidates for reuse will be successful. On the other hand, schemes and methods that anticipate, in the original software design, reuse of the modules with and without modification should produce modules that can be reused successfully and economically. It is not enough to specify the software so that its characteristics are clear, because the requirements of the new system will differ slightly from the original; the software also must be easily modified to meet the new requirement. Thus, precise specification of function and interface and ease of change are essential features of software that is designed to be reusable.

2.2.1 Ada-Related Activities

A description of Ada can be found in reference 2-1.

2.2.1.1 CAMP

CAMP is demonstrating the feasibility and payoffs of reusable Ada parts in a real-time, embedded, mission-critical missile application. The potential benefit to be demonstrated is that software reuse can decrease software cost and increase software quality without sacrificing efficiency.

CAMP has three phases to develop the parts and composition system to aid in the location, understanding, and management of the parts. The first phase, CAMP-1, included a domain analysis over missile flight software systems to identify commonality among these systems. The requirements and architectures of more than 250 software parts were developed. The parts were categorized as simple, complex generic, and schematic. A simple part is one that can be used with tailoring limited to data typing, such as basic data types and vector scalar operations. A complex generic part requires significantly more tailoring, including data types and special operations, such as, navigation and autopilot packages. A schematic part is a blueprint for a class of components, accomplished by a set of construction rules for building application-dependent components given specific requirements. An example of this latter category is a Kalman filter with sparse matrices coupled with a parts composition system. Table 2-4 is a detailed list of parts developed and tested in the program and figure 2-7 shows the parts taxonomy. The specification and architectural design of the parts composition system, the Ada Missile Parts Engineering Expert (AMPEE), was also developed during this phase of the program.

TABLE 2.4. CAMP Parts List

- Missile operation parts
  - Navigation parts
    - Common navigation parts
      - Altitude integration
      - Compute gravitational acceleration
      - Compute gravitational acceleration sin Lat In
      - Compute ground velocity
      - Compute heading
      - Compute rotation increments
      - Compute scalar velocity
      - Update velocity
  - North pointing navigation parts
    - Compute coriolis acceleration
    - Earth relative navigation rotation rate
    - Earth rotation rate
    - Latitude integration
    - Longitude integration
    - Radius of curvature
    - Total platform rotation rates
  - Wander azimuth navigation parts
    - Compute coriolis acceleration
    - Compute earth relative horizontal velocities
    - Compute east velocity
    - Compute latitude
    - Compute latitude using arcain
    - Compute longitude
    - Compute north velocity
    - Compute total angular velocity
    - Compute wander azimuth angle
    - Coriolis acceleration from total rates
    - Earth relative navigation rotation rate
    - Earth rotation rate
    - Radius of curvature
    - Total platform rotation rate
  - Kalman filter common parts
    - Error covariance matrix manager
    - State transition matrix manager
    - State transition and process noise matrixes manager
  - Kalman filter compact H parts
    - Compute Kalman gain
    - Kalman update
    - Sequentially update covariance matrix and state vector
    - Update error covariance matrix
    - Update state vector
  - Kalman filter complicated H parts
    - Compute Kalman gains
    - Kalman update
    - Sequentially update covariance matrix and state vector
    - Update error covariance matrix
    - Update state vector
  - Direction cosine matrix
    - CNE operations
      - Alignment parts
      - CNE initialized from earth position
      - CNE initialized from reference
      - CNE integration
      - CNE from quaternion
      - Reorthonormalize CNE
    - DCM general operations
      - Align DCM matrix
      - DCM initialized from reference
      - DCM trapezoidal integration
      - DCM from Quaternion
      - Frame misalignment
      - Perform rectangular integration of DCM
      - Reorthonormalize DCM
- Guidance and control parts
  - Autopilot
    - Integral plus proportional gain
    - Lateral directional autopilot
    - Pitch autopilot
  - Waypoint steering
    - Compute turn angle and direction
    - Compute turning and non-turning distances
    - Crosstrack and heading error opers.
      - Compute
      - Compute when not turning
      - Compute when turning
    - Distance to current waypoint
    - Steering vector operations
    - Turn test operations
      - Start test
      - Stop test
- Non-guidance control
  - Air data parts
    - Barometric altitude integration
    - Compute dynamic pressure
    - Compute mach
    - Compute outside air temperature
    - Compute pressure ratio
    - Compute speed of sound
  - Fuel control parts
- General operation parts
  - Abstract data structures
    - Bounded FIFO buffer
    - Bounded stack
    - Non-blocking circular buffer
    - Unbounded FIFO buffer
    - Unbounded priority queue
    - Unbounded stack
  - Abstract processes
    - Event-driven sequencer
    - Finite state machine
    - Mealy machine
    - Sequence controller
    - Time-driven sequencer
  - Asynchronous control
    - Aperiodic task shell
    - Continuous task shell
    - Periodic task shell
    - Data-driven task shell
    - Interrupt-driven task shell
  - Communication
    - Message checksum
    - Update exclusion
  - General utility
    - Instruction set test
    - Memory checksum
    - Memory declassification
  - Equipment interface parts
    - Missile radar altimeter
    - Missile radar altimeter with auto power on
    - Bus interface
    - Clock handler
  - Mathematics
    - Coordinate vector matrix algebra
    - Coordinate data types
    - Cross product
    - Matrix matrix multiply
    - Matrix operations
    - Matrix scalar operations
    - Matrix vector multiply
    - Vector operations
    - Vector scalar operations
  - External form conversion
  - General purpose math parts
    - Accumulator
    - Change accumulator
    - Change calculator

TABLE 2.4. CAMP Parts List

- Decrementor
- Extrapolation
- Integrator
- Interpolation
- Lookup table even spacing
- Lookup table uneven spacing
- Mean absolute difference
- Mean value
- Root sum of squares
- Running average sign
- Square root

General vector matrix algebra
- Diagonal full matrix add (unrestricted)
- Diagonal matrix operations
- Diagonal matrix scalar operations
- Dot product operation (restricted)
- Dot product operations (unrestricted)
- Dynamically sparse matrix operations (constrained)
- Dynamically sparse matrix operations (unconstrained)
- Matrix algebra package
- Matrix matrix multiply (restricted)
- Matrix matrix multiply (unrestricted)
- Matrix matrix transpose multiply (restricted)
- Matrix matrix transpose multiply (unrestricted)
- Matrix operations (constrained)
- Matrix operations (unconstrained)
- Matrix vector multiply (restricted)
- Matrix vector multiply (unrestricted)
- Symmetrix full storage matrix operations (constrained)
- Symmetrix full storage matrix operations (unconstrained)
- Symmetric half storage matrix operations
- Vector matrix multiply (restricted)
- Vector matrix multiply (unrestricted)
- Vector operations (constrained)
- Vector operations (unconstrained)
- Vector scalar operations (constrained)
- Vector scalar operations (unconstrained)
- Vector vector transpose multiply (restricted)
- Vector vector transpose multiply (unrestricted)

Geometric operations
- Compute segment and unit normal vector
- Great circle arc length
- Unit normal vector
- Unit radial vector

Polynomial parts
- Chebyshev
- Chebyshev degree operations
- Chebyshev radian operations
- Chebyshev semicircle operations
- Continued fractions
- Continued radian operations
- Fike
- Fike semicircle operations
- General poly nomial
- Hart
- Hart radian operations
- Hastings
- Hastings degree operations
- Hastings radian operations
- Modified Newton-Raphson
- Square Root
- Newton-Raphson
- Square Root
- System Functions
- Base 10 Logarithm
- Base N Logarithm
- Degree Operations
- Radian Operations
- Semicircle Operations
- Square Root
- Taylor Series
- Base 10 Logarithm
- Base N Logarithm
- Degree Operations
- Radian Operations
- Semicircle Operations
- Square Root

Quaternion Operations
- Product of Quaternions
- Quaternion Computer from Euler Angles
- Normalized Quaternion

Signal Processing Parts
- Absolute Limiter
- Absolute Limiter with Flag
- General First Order Filter
- Lower Limiter
- Second Order Filter
- Tustin Integrator with Asymmetrix Limit
- Tustin Integrator with Limit
- Tustin Lag Filter
- Tustin Lead-Lag Filter
- Upper Limiter
- Upper Lower Limiter

Standard Trig
- Sine
- Cosine
- Tangent
- Sine_Cosine
- Arcsine
- Arccosine
- Arctangent
- Arcsine_Arccosine

Static Sparse Matrix

Unit Conversions

Data Type and Object Pars

Data Types
- Autopilot Data Types
- Basic Data Type
- Kalman Filter Data Types

Constants
- Conversion Factors
- WGS72 Ellipsoid Data (Engineering)
- WGS72 Ellipsoid Data (Metric)
- WGS72 Ellipsoid Data (Unitless)
- Universal Constants

CAMP-2 included the implementation, testing, and cataloging of the missile parts. The AMPEE system was mechanized and used to implement software for an actual missile application, named the 11th missile. This phase of CAMP also developed a set of armament electronics (Armonics) and benchmarks.

The Armonics benchmarks allow the user to compare and select appropriate functions for a given application trading off accuracy versus processing time, for example, a cosine routine. Other tests will measure compiler capabilities in terms of correctness, size, and speed of the generated code.

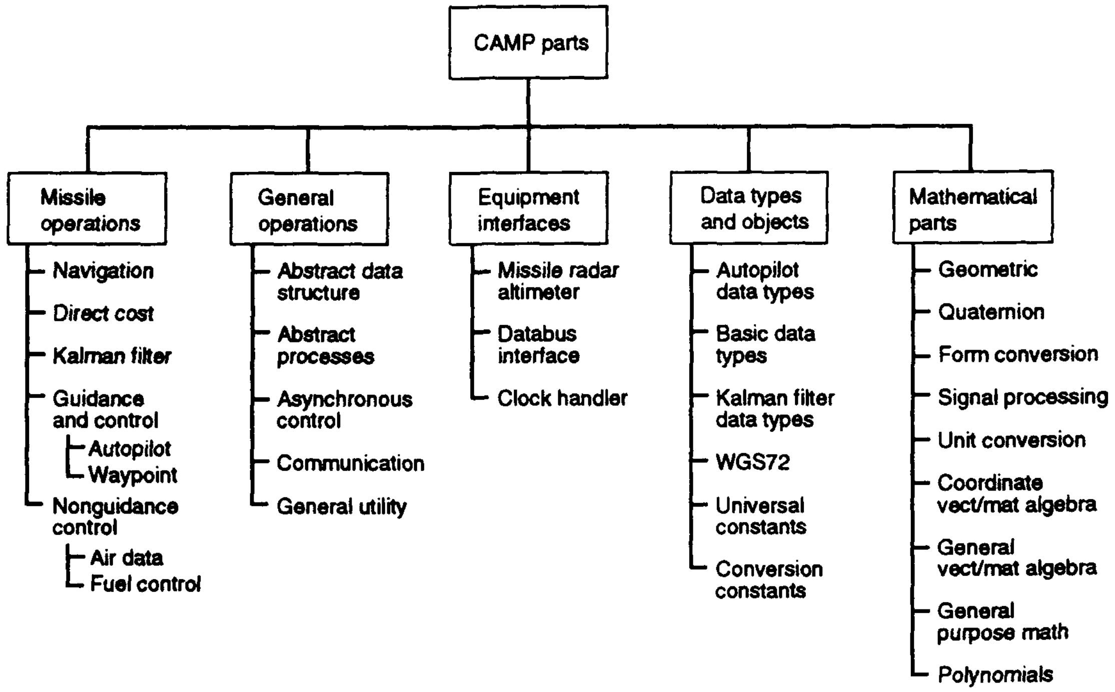


Figure 2-7. CAMP Parts Taxonomy

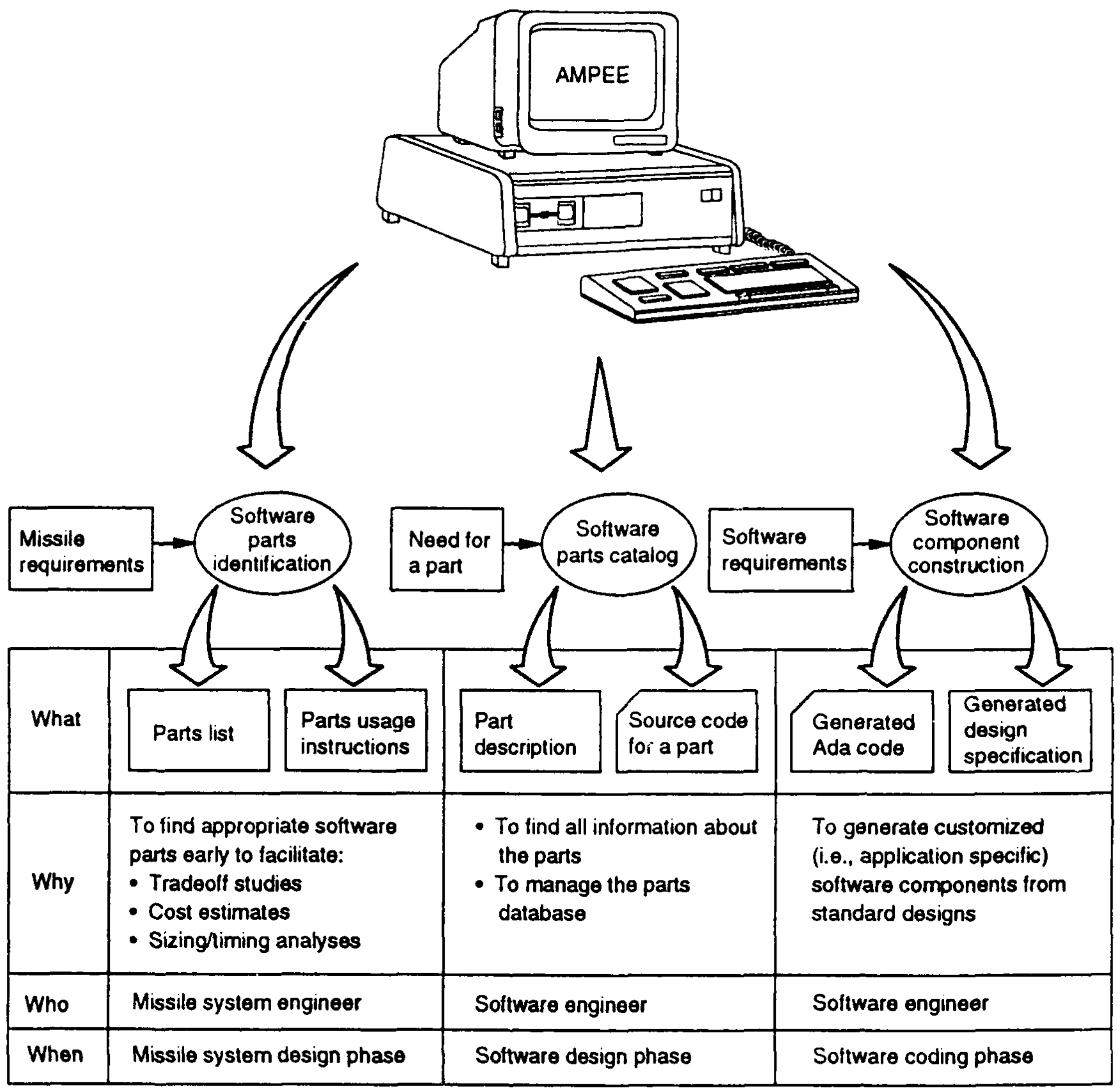
The 11th missile application used the parts generated from the domain analysis of 10 missiles. The goal was to measure the effectiveness of the CAMP parts and AMPEE system. The requirements were derived from an existing application, which was implemented in JOVIAL. Results showed that 21% of the parts were directly reused, 3% were used with modifications, and the remainder were not used. Of the parts not used, 52% represented duplicated functions, 40% were not applicable, and the remaining 8% were incompatible with the application.

The AMPEE system, figure 2-8, provides the user with three major facilities: software parts identification, software parts catalog, and software component construction. During the missile design phase, the parts identification function aids the system engineer in mapping system requirements to applicable software parts. As an example, if a north-pointing navigation system is required, identification of all appropriate parts and instruction on their use is provided. After a part is tentatively identified, the parts catalog is queried to obtain detailed information. Table 2-5 lists the attributes of the parts in the catalog. The component constructor facilitates the generation of Ada application software from complex generic and schematic parts.

The CAMP-3 effort is doing further component exploration and construction and will update the catalog. The library will also be maintained during this phase and catalogs will be distributed to users.

2.2.1.2 STARS

The STARS program was started in 1983 by DoD. Various Ada shadow projects and foundation contracts were started in 1986, and three cooperating prime contracts were awarded in August 1988. The purpose of the shadow contracts was to investigate the technical issues when applying Ada to various systems. The foundation contracts purpose was to build up a database of applications for future reuse studies and applications. Boeing, IBM Federal Systems, and Unisys are on multiyear, five-period, task-oriented contracts.



What	Parts list; Parts usage instructions	Part description; Source code for a part	Generated Ada code; Generated design specification
Why	To find appropriate software parts early to facilitate: • Tradeoff studies • Cost estimates • Sizing/timing analyses	• To find all information about the parts • To manage the parts database	To generate customized (i.e., application specific) software components from standard designs
Who	Missile system engineer	Software engineer	Software engineer
When	Missile system design phase	Software design phase	Software coding phase

Figure 2-8. The AMPEE System

Tasks to be performed in the five periods include the following:

a. Establish an Ada baseline. In this period, a baseline tool set and environment, an environment evolution strategy, and a reusability strategy will be developed.

b. Explore *reusability* benefits. In this period, the foundation contracts will supply reusable parts to the cooperating primes, who will explore the benefits and problems of reusability. (See STARS reusability guidebook.)

c. Apply prototype STARS environments. A prototype software development environment will be delivered during this phase.

d. Integrate STARS-developed technologies. Various tools and methods developed over the first three phases will be integrated into a fully operational, automated development environment.

e. Develop training programs and application plans. Plans for installing and using the environment will be developed. Then, a plan for applying the system in major DoD programs will be developed.

Table 2-5. AMPEE Parts Attributes

Required attributes	Recommended attributes
• Part number	• Keywords
• Revision number	• Project usage
• Part name	• Remarks
• Taxonometric category	• Design issues
• Functional abstract	• Requirements documentation
• Class	• Design documentation
• Mode	• Developed for
• Last change date of entry	• Location of source code
• Development date	• Access notes
• Developer	• Revision notes
• Development status	• Implements
• Government security classification of part	• Implemented by
• Government security classification of entry	• Used to build
• Corporate sensitivity level of part	• Built from
• Corporate sensitivity level of entry	• Sample usage
• Withs	• Restrictions
• Withheld by	• Hardware dependencies
	• Source size/complexity
	• Timing
	• Fixed object code size
	• Accuracy

STARS takes advantage of the inherent features of the Ada language and is building a software environment that will permit software programs at the applications source code level to be machine independent. Furthermore, at the Naval Research Laboratory, the STARS program is developing an object-oriented repository or library to promote software reusability.

STARS will use the *software-first* concept, which advocates developing software to a mature state before specifying the hardware design. In the software-first concept shown in figure 2-9, activities within each phase refine the product of the phase in a cyclic fashion, with hardware decisions being delayed as long as possible. Each product flows on to the next phase when it is deemed sufficiently detailed for further development, allowing iterative development within and between phases, as opposed to the traditional waterfall life cycle in which products flow forward only when a phase is complete.

The STARS program is also concentrating on the front-end development period, where rapidly prototyping major portions of a major software system can be done using the reusable building blocks in a repository database. Rapid prototyping helps to reduce risk (schedule and cost) and supplies needed *lessons learned* from earlier development programs.

The STARS program is investigating a work team concept. In this concept, a system architect heads a group of chief programmers and each chief programmer leads a small group of programmers. In very large software systems, several system architects may report to a chief system architect. This concept is not new and is now being adopted by various groups.

Earlier STARS work (1986 to 1987) produced a reusability guidebook (ref. 2-2). It envisioned an inventory of reusable parts in a computerized library system accessible by projects developing software. The

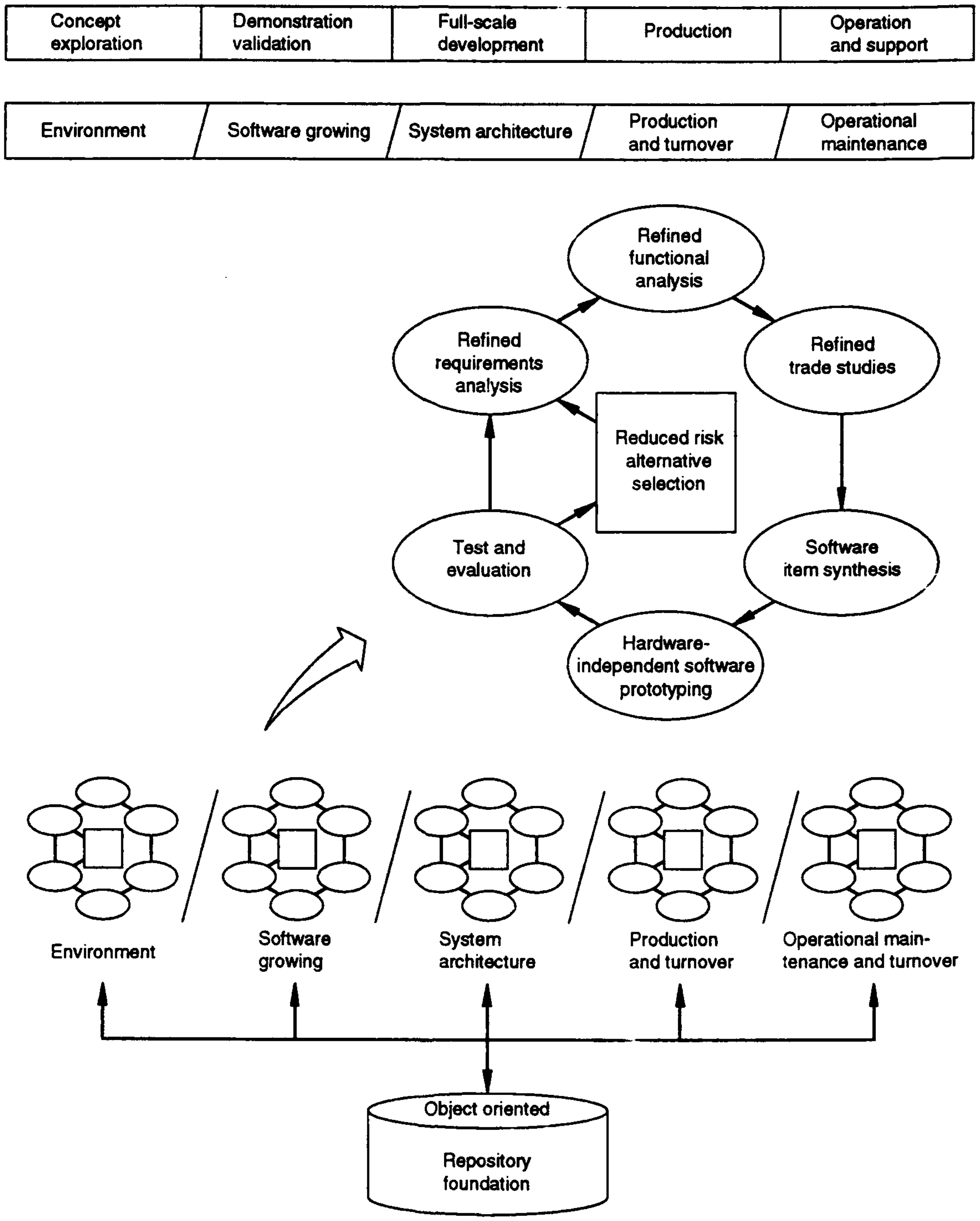


Figure 2-9. Software-First Acquisition Cycle

guidebook examined characteristics of software modules that have a bearing on a module's suitability for reuse. The following is a list of the characteristics and their meanings (if the meaning isn't obvious):

a. *Balance between generality and specificity.*

b. *Complete.*

c. *Efficiency.*

d. *Environment independent.*

f. *Highly cohesive.* The part contains all relevant information within itself.

g. *Information hiding.* The part hides the implementation details.

h. *Loosely coupled.* The part has minimum dependencies on other parts.

i. *Localization.* Information within the part has a logical grouping.

j. *Modifiable.*

k. *Primitive.* The part can be reused without any knowledge of the underlying implementation.

l. *Protection against incorrect use.* The part enforces external assumptions through a rigid part interface.

m. *Reliable.*

n. *Simplicity of interface.*

o. *Understandable.*

p. *Uniform.* The part follows the conventions of the other parts.

It was recognized in the STARS effort that, even though it is difficult to derive, a set of metrics is needed to help in the evaluation of software modules. Following is the set derived (many were taken from reference 2-3). Some are general metrics; some are Ada specific (those will be noted in the list).

a. *Application independence.* Is application-specific information passed as subprogram or task entry parameters rather than contained within the subprogram or task? (Ada specific.)
Is application-specific information passed as generic actual parameters to generic subprograms and/or packages rather than being contained within the program units themselves? (Ada specific.)

b. *Document accessibility.* Are subprogram, package, task, and generic program units written following a standard format? (Ada specific.)

c. *Generality.* Are default parameters used in subprogram and/or task entry formal parameter parts to generalize the context of the subprogram? (Ada specific.)
Are default generic actual parameters used to generalize the context of the subprogram? (Ada specific.)
Have generics and generic parameters been used to generalize the context? (Ada specific.)

d. *Modularity.* Have Ada decomposition-related capabilities been used? (Ada specific.)

e. *Augmentability.* Have package/subprogram/task/generics program unit specifications been separated from their corresponding bodies? (Ada specific.)
Do subprogram/task bodies have one normal exit and a grouped set of abnormal exits? (Ada specific.)
Are all subprogram and task entry default parameters grouped at the end of corresponding parameter specifications? (Ada specific.)
Are use-clauses avoided? (Ada specific.)
Have generics been used to the fullest extent possible? (Ada specific.)

f. *Trust.* Each part is assigned a value in a number of areas related to trust. The resulting trust vector can then be used to determine the trustworthiness of the part when it is considered for reuse.

g. *Completeness.* Which of the following constituents of the part are present in the library: abstract, requirement specification, functional specification, design, algorithm description, source code, object code, test specification, test code, test data, manuals, and training materials?

h. *Use.* The use metric indicates how widely used the part is. It shows the number of times retrieved, used with modification, used without modification, and retrieved but not used.

i. *Design for reuse.* This measures the consideration given in design to the reuse of the module. Contractors are producing modules to be stored in a reuse library at the Naval Research Laboratory in Washington, D.C.

2.2.1.3 GRACE

GRACE is a set of Ada programs implementing commonly used data structures. GRACE is sold and maintained by EVB Software Engineering, Inc. The set of programs is based on Grady Booch's taxonomy of program families (ref. 2-4). GRACE implements approximately 30 basic data structures with variations, such as bounded and unbounded requests, for a total of 275 distinct components.

GRACE is written entirely in Ada and is self-contained. It is machine and operating system independent and requires only a validated Ada compiler. The 275 components total 500,000 lines of Ada source code. Each component contains the requirements and design documentation, the source code, test programs, and a bibliography. A component is between 40 and 125 pages in length; 70 pages is typical. The generic Ada package is between 400 and 1,000 lines of Ada code.

The variations of a particular data structure are determined by the way the data structure is merged. A basic distinction is between programs to be used in a single-process environment and a multiprocess environment. Following is a brief overview of different forms for the data structures.

a. *Sequential/parallel.* A *sequential* component behaves properly in a nonconcurrent environment. A *parallel* component is designed to minimize the chances of deadlock or starvation in a multiprocess environment. Parallel components are *protected, multiple, guarded,* or *multi-guarded* depending on the type of multiaccess protection inherent in the component.

b. *Controlled/uncontrolled.* Controlled software has state information concerning the data structure that will be protected from corruption caused by simultaneous access by concurrent processes. Uncontrolled software either maintains no state information or does not protect it from corruption by concurrent processes.

c. *Bounded/unbounded/limited.* A *bounded* component has no dynamic allocation. An *unbounded* component has a data structure that may dynamically grow or shrink. An unbounded structure may have an upper limit, in which case it is called *limited.*

d. *Managed/unmanaged.* If a component manages its own memory it is classified as *managed.* If, however, it depends on the operating system's memory management system to allocate and deallocate memory space or nodes in the data structure, it is classified as an *unmanaged* component.

e. *Iterator/constructive iterator.* If the component provides a means for systematically transversing the data structure, then the component is said to be an *iterator* component. If changes are allowed to be made to nodes during iteration, the component is called a *constructive* iterator.

f. *Operation/object.* This distinction has to do with the method of controlling concurrent operations. In operation-based concurrency, the concurrency control is based on the software accessing an object (i.e., the code within the operation is sequentialized to prevent any possible problems associated with concurrent operations). In object-based concurrency, the control of concurrency is associated with the object itself. The code accessing the object is not sequentialized.

g. *Protected and Multiple/Guarded and Multiguarded.* If the concurrency control is based on the component (rather than the object), it can be handled completely by the component (by sequentializing

the code) or the component can allow the user to implement the control using semaphores. The former is called *operation protected*; the latter is called *operation guarded.*
If the protection is provided on an object basis rather than a task basis, then sequential access to an object is called *object protected.* If the user can control this access using semaphores then the term used is *object guarded.*

Access to the state of an object is either read only, called a *selecto*; or read and write, called a *constructor.* Mechanisms are provided for multiple, simultaneous selector operations to occur. When a constructor operation is scheduled to occur, all currently executing selector operations are allowed to complete, all additional selector operations are blocked, and then the constructor operation executes to completion. The same distinction is made as stated before. If the control is based on the component, it is called *multiple.* If the control is based on an object, it is called *object multiple.*

Following is the component listing. Most of the components are also divided into *bounded* or *unbounded*, *iterator* or not, and *managed* or not.

Ring structure	Queue
Set	BST
Singly linked	Circular doubly linked
Stack	Dequeue
String	Directed graph
Undirected graph	Doubly linked list
Balanced tree	Hash table
Binary tree	Heap
Binary search tree	Map
Circular doubly linked list	Multiset
Directed graph	Priority queue
Discrete set	Queue
Doubly linked list	Singly linked list
List	Stack
Matrix	Weighted directed graph

2.2.1.4 Ada-Based Integrated Control System

The Ada-Based Integrated Control System (ABICS) was a flight research program to apply Ada programming language to operational flight programs of flight-critical and mission-critical systems. Flight testing was accomplished on an F-15 test aircraft. The program consisted of three phases with successive stages of integration. Phase I of the program converted the F-15 dual command augmentation system into a digital flight control mechanization. In phase II, an integrated flight/fire control system was mechanized with these coupled control modes: air-to-ground gunnery, air-to-air gunnery, and bombing. A dual strap-down, laser gyro system was integrated into the system in phase III of the effort. Flight testing was done in all three phases.

The primary software goals of the program were to assess the applicability of Ada in an integrated control system application and gather data to generate information on computer resource expansion and productivity. A secondary goal was to examine reusability potential of the developed software.

For phase I, the development process started from a verified and validated set of control and redundancy management algorithms. The situation was the same for phase II, except that the new software modules were developed. During phase III, a portion of the software was developed. For the cases where reusable algorithms were applied, an order of magnitude improvement in productivity measured in man-hours per word was observed. This was in comparison to the original development. Another factor contributing to the productivity gains was the reuse of a set of test cases and conditions.

Reusability was assessed for two applications: (1) an Ada-compatible computer on the F-15 and (2) a different Ada-compatible computer on a different fighter aircraft. Phase I results show that 75% of the modules could be reused for the first application, and the redundancy management portion could be reused for the second application. The reusable modules resulting from phase II are in table 2-6. More than 80% of the software could be reused for condition 1 applications, while 50% could meet condition 2 applications. For phase III, three more modules were developed: sensor processing, velocity and attitude computation, and navigation computation. The latter module was mechanized in Ada while the first two modules were in assembly to meet execution time requirements. The first module is sensor specific and cannot be directly reused, and the other two modules can be reusable for both condition 1 and 2 applications.

Although the ABICS program did not specifically target reusability as a primary goal, it demonstrated that reuse is achievable in the areas of design, code, and testing.

Table 2-6. ABICS Phase III Reusable Modules

Module	Reusable in other F-15 applications	Reusable in other fighter aircraft
Air-air gunnery (AAG)		
• Executive	No	No
• OBS	Yes	Most
• Navigation processor	Yes	No
• Radar preprocessor	Yes	No
• Aircraft state estimator	Yes	Yes
• Kalman filter	Yes	Yes
• Target relative state estimator	Yes	Yes
• Gunnery equations	Yes	Yes
• Gun scoring	Yes	Yes
• Coupler	Yes	Conditional/most
• Displays	Yes	No
• IFIM	Most	No
• BIT	Most	No
Bombing (BMG)		
• Target designator	Yes	No
• BMG equations	Yes	Yes
• China Lake	Yes	Yes

2.2.2 Other Activities

2.2.2.1 Software Cost Reduction

As noted in section 2.1.1, most of the current, normal software for guidance and control systems cannot be reused because reuse was not a consideration when the software was produced. For software to be reusable, certain design and programming disciplines must be followed while the software is being produced. This is not a great burden on the software producers, however, because the same good design principles that allow the software to be designed and built rigorously are precisely the ones that allow the software to be reused easily. In particular, software development methodologies that enhance the software maintainability are particularly useful in building and cataloging software modules for reuse. A methodology that encourages software development for reuse has been built on Parnas's idea of using information hiding to design software systems that can be expanded (or contracted) by the addition (or deletion) of well-defined software modules (refs. 2-5 and 2-6).

The Software Cost Reduction (SCR) project was conducted at the Naval Research Laboratory from 1977 through 1988. Its goal was to develop and refine software engineering techniques and to transition them into use in the development of Navy software systems. To aid in the transition and to provide model documents that system designers and developers could follow, an existing Navy real-time system (the A-7E onboard flight software) was chosen as an example application for the new methodology.

The philosophy of the SCR methodology is the specification of what product to work on in each phase of the software design and development, what the rules are for the content and completion of the product, who should work on the product, and who should use the product. The major software engineering principles underlying the SCR methodology are precise specifications, design through documentation (ref. 2-7), abstract interfaces [DIM], modularization according to information hiding principles, and process synchronization. Specifically, the project developed a new software requirements methodology (refs. 2-8 through 2-10), a new design review technology ([DIM] ref 2-11), precise guidelines for decomposition into software modules according to information-hiding principles (refs. 2-5, 2-6, 2-12 and 2-13), specification techniques for information-hiding modules ([DIM] refs. and 2-15), a practical method for using abstract data types (ref. 2-16), a method to enhance software portability (ref. 2-17), a new control construct (ref. 2-18), and new real-time process synchronization primitives (ref. 2-19).

The following description of the applicability of SCR methodology in software reuse has been summarized from reference 2-17.

The candidate unit of software for reuse is the *module*. In this terminology, a module is a work assignment for a programmer or programmer team. Each module consists of a group of closely related programs. The module structure of a system is the decomposition of the system into modules and, for each, the assumptions that the teams responsible for other modules are allowed to make about it. Each set of assumptions is called the *interface* to the module.

The overall goal of the decomposition into modules is reduction of software development and maintenance cost by allowing modules to be designed, implemented, and revised independently. These properties are synonymous with reusability. Specific goals of the module decomposition are:

a. Each module's structure should be simple enough that it can be understood fully.
b. It should be possible to change the implementation of one module without knowledge of the implementation of other modules and without affecting the behavior of other modules.
c. The ease of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed. It should be possible to make likely changes without changing any module interfaces; less likely changes may involve interface changes, but only for modules that are small and not widely used. Only very unlikely changes should require changes in the interfaces of widely used modules. There should be few widely used interfaces.
d. It should be possible to make a major software change as a set of independent changes to individual modules; in other words, except for interface changes, programmers changing the individual modules

should not need to communicate. If the interfaces of the modules are not revised, it should be possible to run and test any combination of old and new module versions.

Software that meets the goals above (a through d) is composed of many small modules organized into a tree-structured hierarchy; each nonterminal node in the tree represents a module that is composed of the modules represented by its descendants. The hierarchy is intended to achieve the following additional goals.

e. A software engineer should be able to understand the responsibility of a module without understanding the module's internal design.

f. A reader with a well-defined concern should be able to easily identify the relevant modules without studying irrelevant modules. This implies that the reader will be able to distinguish relevant modules from irrelevant modules without looking at their internal structure.

The module structure is based on the decomposition criterion known as information hiding (ref. 2-6). According to this principle, system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change. For example, every data structure is private to one module; it may be directly accessed by one or more programs within the module, but not by programs outside the module. Any other program that requires information stored in a module's data structures must obtain it by calling programs on the module's interface.

Three ways to describe a module structure based on information hiding are (1) by the *roles* played by the individual modules in the overall system operation, (2) by the *secrets* associated with each module, and (3) by the *facilities* provided by each module. A module guide describes the module structure by characterizing each module's secrets. Where it would be useful, the module guide also includes a brief description of the role of the module. Each module has a precise abstract specification that tells exactly what the module does. The module guide tells which module(s) will require a change. The module specification tells how to use that module. The issues of developing a software catalog or module guide for a system are discussed in greater detail in references 2-13 and 2-20.

The specification of an abstract interface in the module guide or in the catalog entry that indexes the modules should have the following properties:

a. It must not disclose any of the changeable aspects of the module.

b. It must present a concise description of the facilities available from the module, in terms of effects that are observable to the user.

c. It should be divided into sections and formatted so that a reader unfamiliar with the module is able to find a piece of information without having to study the entire interface specification; in other words, it should serve the quick-reference reader as well as the first-time reader.

d. It should not provide unneccessary duplication of information which would make using and maintaining the document more difficult.

The SCR organization for specification of abstract interfaces (ref. 2-20) is designed to achieve these properties.

Most of the problems of reusing software modules are problems of scale. Problems that are insignificant for small programs can be overwhelming for large ones. With a library of 20 programs, there is no real problem with reuse. With 20,000 programs, the problem is obvious. This section shows how software that is structured in a well-defined and well-documented fashion lends itself to reuse, and, in particular, how the last four problems mentioned in section 2.1.1 are overcome.

a. Structuring software using information hiding demands precise specification. It is impossible to use the principle without providing a substitute for the information that is hidden. The module specifications remove this cause for software not being reused.
b. The fact that all such software is designed to be easy to change increases the likelihood that it will be reused. Further, information hiding leads to separation of concerns. Device-dependent code will be in one module, physical models somewhere else. This makes it easier to reuse the device dependent code with a different physical model, or to reuse the physical models with different devices.
c. A module guide of the kind described serves as a software catalog. Users can characterize the software they seek by the same secrets that characterize the software described in the catalog. The hierarchical nature of the catalog lends itself to rapid convergence to that part of the existing system that will most likely provide the needed software.
d. Such software should be designed to be flexible, not general[4]. Software should not be designed to be used without change in many situations; that would be inefficient. Instead, it should be designed so that it is easy to adapt to a new situation.

Information hiding and abstraction are two sides of the same coin. The value of abstraction has always been that results developed in the implementation of an abstraction may be reused for any valid model of that abstraction. By developing and cataloging abstractions, we greatly increase the likelihood that some of the software will be reused.

2.2.2.2 Software Productivity Consortium

The Software Productivity Consortium (SPC) was established in 1985 to investigate methods for increasing the productivity of software engineers developing Ada-based embedded systems. Fourteen charter members of the consortium are:

Boeing	Martin Marietta
Ford Aerospace	McDonnell Douglas
General Dynamics	Northrop Corporation
Grumman Aerospace	Rockwell International
Harris Corporation	TRW
Hughes	United Technologies
Lockheed	Vetro Corporation

The objective of SPC is to:

a. Define a methodology that supports reuse and rapid modeling, including prototyping.
b. Automate the methodology as much as possible in a set of tools.
c. Provide these tools on a variety of platforms, integrated with existing tools that are already available commercially.
d. Apply and validate the methodology internally.

[4] *The distinction is discussed in reference 2-5.*

e. Teach the methodology and supporting tools to the member companies.

f. Evolve the methodology and tools to meet member company requirements.

g. Transfer and support the associated tools as they evolve.

SPC is developing rapid prototype methods and tools, reuse methods and libraries, and a synthesis method that will enable the consortium and members to exploit prototyping tools and reuse methods.

SPC defines a prototype as an executable model that approximates the end product and is relatively quick, easy, and inexpensive to build and analyze. The prototyping approach is one that can be used anytime in the development life cycle where the technical risk is high. Rapid prototyping composition implies use of existing (reusable) parts or new models. The reusable parts are held in a reuse library which grows in number over time. Reusable parts from the library often require modification to meet each application need, but, as the library grows in size, users will be able to find pieces that will better approximate the final end product. SPC is developing a process called synthesis which relies on the prototype software modules for reuse.

Inherent in synthesis is the idea that, at any point in system development, the existing models approximate what the final system must ultimately be. Each model is iteratively enhanced until the approximation becomes close enough to satisfy requirements. This is a very flexible approach allowing for experimentation as required. For example, early in the development, a quick prototype might be needed to validate the user interface of the system. Approximate application, design, and implementation models might be suitable for that purpose, assuming the approximations were relatively good for the user interface aspects of the system, but poorer for other aspects.

The basic methodology used for prototyping and reuse is described in reference 2-21. The major ideas used are *program families* as defined in references 2-5 and 2-22, *information-hiding modules* (refs. 2-6 and 2-13), and *abstract interfaces* (ref. 2-23).

Because the new system is built from old parts, much of the validation has already been done. Evolving models of the new system are validated throughout the development process to ensure that the process converges. Sometimes the models will be executable; that is, prototypes.

The SPC synthesis concept can be contrasted to the classic waterfall model and to some current examples of application generators that are completely specific to a target application, not extendible to other applications, and do not allow intervention and customization at all three model levels. Maintaining a system's view of system and software development, the SPC tools and toolsets will be built as parts of a system solution, in contrast with many current tools, and that operate well in limited areas but do not integrate well with overall development methods.

The SPC is working to provide the methodology and supporting structures for a powerful synthesis process. They will provide member companies with the internal mechanisms and tools to do synthesis. SPC members provide their own proprietary domain expertise in sectors pertinent to their respective business interests. Their creation of a synthesis capability for domains including parts libraries may be a multiyear commitment, but the investment is necessary to maintain competitiveness in some domains. The reuse library and domain-specific automated aids become corporate assets.

Synthesis works in part by setting an appropriate context for everyone in a project. Each contributor works in familiar professional territory with each context having representation based on the same data.

Figure 2-10 relates the three major subprocesses of synthesis: domain analysis, system development and evolution, and reengineering for reuse. They are connected mainly by reusable work products and the use in system development and evolution of domain-specific tools developed by domain analysis. Portions of the overall process repeat when the subprocesses provide new information that necessitates revising an earlier decision or adding additional definition of the end product.

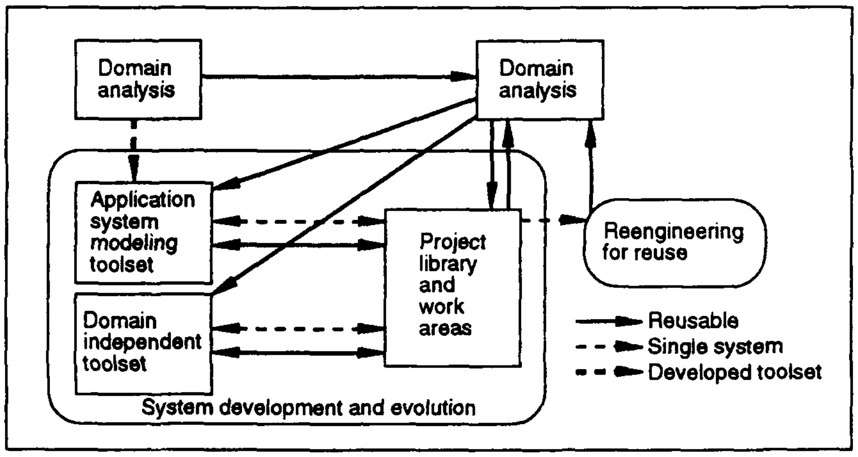

Domain analysis
Domain analysis
Application system modeling toolset
Project library and work areas
Reengineering for reuse
Domain independent toolset
System development and evolution
Reusable
Single system
Developed toolset

Figure 2-10. Three Major Synthesis Subprocesses

First, synthesis begins with domain analysis, an analysis and engineering activity sometimes called domain engineering. To achieve the capability for synthesis in an applic. ion domain, the domain must be characterized in application terms and by canonical designs or architectures that capture how systems in the area could reasonably be built. Domain analysis may be an activity with the primary goal generating a reusable domain or it may be an incremental activity where successive projects build upon previously produced materials.

Second, the system development and evolution of an application using the domain libraries and tools have three main activities: (1) create a model that approxima'es the new application, (2) map from the application model to a software design model that approximates a valid design, and (3) map from the software design model to executable model/code that approximates a valid implementation. These steps are performed iteratively, composing and adapting the models to become even better approximations of the desired new system. At each step, appropriate validation, verification, and assessment are performed.

Third, reengineering for reuse is feasible where single system software exists that has portions that adapted into reusable parts. It includes extraction, generalization, standardization, and certification. The results are cataloged and stored in reuse libraries.

2.2.2.3 CARE

CARE is a joint University of Maryland and ITALSIEL, S.P.A. project (ref. 2-24). Its purpose is to make more software components reusable by reengineering existing software. The software is analyzed in two steps: first, the process identifies the reusable components; then, it classifies the components according to a reuse specification. The software components are then stored, along with a specification in a software repository for later use.

An initial prototype system is expected to be completed that, with the help of a domain expert, will:

a. Extract and select the components from a set of programs.

b. Measure their reusability and frequency of reuse.

c. Restructure the selection according to those measurements.

d. Correlate reuse with a measure of program complexity.

e. The initial prototype can be used on Ada and C programs. It is being implemented on a SUN workstation running UNIX[5].

The major concern of the CARE developers is the reuse of existing software components (as opposed to the thrust of the SCR project, which is to structure software when it is designed so that it can be reused). The process concentrates on defining and then measuring characteristics that make components more reusable. The selected components are then made more reusable by writing a set of specifications.

Two domains are defined for a software engineering project that is concentrating on reuse. The first is the *business domain*, which comprises those activities that are related to producing the business software that is delivered to the customer. The second is called the *component factory*, which comprises those activities that are related to the implementation and maintenance of the reusable components, the factory software. This activity includes the tailoring of the reusable components into business software.

A closed waterfall life cycle model is used for the business domain; it is closed to indicate the evolution of specifications and design after the software is released. Between the design and component integration phases of the model, specifications are sent to the software factory and components are sent from the factory back for integration. An important consideration is that the business domain programmers are not allowed to make any change to the reusable component; in fact, details of the actual implementation of the module are not available to the programmer.

The closed waterfall model used for the factory software is somewhat different because the emphasis is on a component lookup based on the specification received from the business domain. If the specified component is not available in the inventory, it is generated and sent back to the business domain. The component is then generalized and stored back in the factory inventory.

The emphasis of CARE is on reengineering existing systems for use in new systems. The systems use the software reuse framework defined in ref. 2-25. The framework defines dimensions in *environment, process*, and *product*. The *environment* dimensions include application stability, process stability, and personnel stability. The *process* dimensions include the time of activities, the type of modification, and the modification mechanism. The *product* dimensions include the type, usability, and quality of the object.

CARE has quantified the notions of small program size, simple structure, good documentation, and suitable language into a set of parameters to measure components within the software factory. These parameters identify reuse components. After identification, the components are qualified by understanding their meaning well enough to write a specification for each.

In the first phase of component identification, the candidate modules (or components) are selected by an automated process using a set of measures that measure the likelihood that a module is a good candidate for reuse. To be selected, a module must have a simple structure, small size, excellent documentation, and be written in a suitable programming language (currently Ada or C). After the initial selection, components are further qualified by applying stricter requirements.

[5] *UNIX is a trademark of AT&T.*

The initial selection criteria are more precisely defined as compactness and independence (for which measures such as cyclomatic complexity, software science measures, data bindings, package visibility, and component access can be used) and frequency of reuse (an observable property of existing systems). In addition, a set of exogenous constants, which are not really constants, are manipulated to set boundaries as the environment is better defined.

Although the measures for selection defined in reference 2-24 are currently based on C or Ada, they are general enough to be instructive. After a subprogram map tree is developed from the source code, two sets of measures are used, μ for general measures and $\surd$ for reuse-specific measures. The current measures used in the prototype are as follows (a more detailed description is contained in reference 2-24):

a. Data types measure: μ dt measures the number of nesting levels between the subprogram defining a particular data type and the subprogram using that data type.
b. Data bindings measure: μ db measures the communication path between subprograms in the form of a triple (A,x,B), where A is the subprogram defining the variable x, and B is the subprogram referencing the variable x.
c. Program size is measured by the number of lines of source code μ LOC and by the Halstead length μ N [MET].
d. Program structure is measured by the number of subprograms and links and by the McCabe cyclomatic complexity measure, μcyc, of the flow graph representing the program (ref. 2-3).
e. Programming language is measured by the Halstead language level indicator, μT (ref. 2-3).
f. Program documentation is a subjective rating with a heavy emphasis on the documentation of the interface.
g. Program reliability can be derived from expected and reported errors in the module.
h. Reuse frequency ((Co)) is the ratio of the number of calls to component o to the total number of calls in some collection of programs.
i. Reuse specific frequency (s(Co)) is the ratio of the number of calls to component o to the number of calls to a particular component in the environment.

After a component module is selected per the previously defined measures, it is *qualified* by writing functional and interface specifications (see reference 2-20 for a way of specifying interfaces) for the module. From these specifications, the component is *classified* per given component taxonomy. The specifications and the classification are stored with the component in the software repository as its *reuse-oriented specification.*

The components are classified and stored in a data bank that is indexed by an entity relationship model, <Object, Function, Medium>. The attributes are not completely defined in reference 2-24, but generally can be divided between functional and environmental attributes. Some examples, taken from reference 2-24, suffice to define the general nature of functional attributes: <Entity 1, Add, Storage>, <Ent1.Attribute1, Modify, Workstation>, <Ent1.Attribute2, Decode, Stack>, and <Entity 2, Compare, Queue>. The environment attributes define the environment external to the component: that is, the external interfaces, called ports, the functional area (the name of the function in the business domain), and the operating environment.

2.2.2.4 Object-Oriented Programming

As previously noted, the major roadblock to software reuse is when the software is not originally designed for reuse. For software to be reusable, certain design and programming disciplines must be followed during software production. But the design and programming disciplines that aid in software reuse are also those that help in the proper construction and documentation of software in the first place. Hence, there is no need to change the way good designers and programmers build systems to enhance software reusability. Two key elements in good software design are information hiding (ref. 2-6) (section 2.2.2.1) and data abstraction (refs. 2-14 and 2-23). These elements allow details to be specified in only one place in the system and hidden from the rest of the software. Thus, software can be modified and reused more easily.

Abstract data types are used to hide the actual implementation of data and to provide a program that can deal with different types of data without having to specify and account for all the different types during program or algorithm design and construction. The system interprets and mediates the data exchange to ensure compatibility and consistency. An abstract data type describes a class of objects characterized by the operations that can be performed on them and by the abstract properties of such operations. Abstract data types are obviously useful in all types of software specification and design, and they form the basis of *object-oriented* systems. One view of object-oriented design is that it is the construction of software systems as structured collections of abstract data-type implementations (ref. 2-26).

This view emphasizes the applicability of object-oriented programming to reusability. As with other reuse schemes, the literal use of object-oriented programming requires that the software to be reused be constructed using an object-oriented language. (See, for example, references 2-27, 2-28, 2-29, 2-30, and 2-31). The module must also be reused in a system using an object-oriented language.

Extensions to the idea of *abstract data type* are the ideas of *inheritance* and *client*. These are the major features of object-oriented systems. One abstract data type class can be an heir to another class (or to other classes in a system that allow multiple inheritance) and inherit all the features of the class. As an example, a binary search tree data class may be an heir to a table data class. Then the binary search tree data class would automatically have all the features of the table data class. Client relationships allow the specification of different features for the manipulation of objects.

Following is an example taken from ref. 2-26 using the notation of reference 2-31. Class A is said to be a client of class B if A contains a declaration of the form bb:B. In this example, A may manipulate bb only through the specifications contained in B. The specification may contain both attributes (data items associated with objects of type B) and routines (operations for accessing or changing these items).

The using of the inheritance and client features of an object-oriented system can isolate (or hide) the portions of software that are likely to change. This increases the likelihood that a module can be reused without change, and, in those cases where changes must be made, they are easier to identify and to accomplish.

For a fuller treatment of the use of object-oriented design for software reuse, see reference 2-26. It contains an instructive example of restructuring a system using object-oriented design to make the modules more reusable.

2.2.2.5 ESPRIT

ESPRIT is a European consortium doing experimental research in software technology. Most of the projects within ESPRIT concentrate on the use of formal methods and transformations. While reusability is a concern in most of the projects, it is a major focus of only one, REPLAY.

2.2.2.6 Formal Methods

The focus of most projects within ESPRIT is on the use of formal (in the mathematical sense) language and methods in developing software. Following is a list of ESPRIT projects with a short description of the project, a set of key words, and the participating organizations and their country code.

GRASPIN: Personal Workstation for Incremental Graphical Specification and Formal Implementation of Nonsequential System
KEY WORDS: formal and informal techniques in the development process, algebraic specification, ADT, Modula-2, Ada.
PARTICIPANTS: GMD and SIEMENS (D), OLIVETTIC (I)

PROSPECTRA: Program Development by Specification and Transformation
KEY WORDS: methodology, Knuth-Bendix, specification, transformation, Ada-AnnA.
PARTICIPANTS: University Bremen, Saarland, Dortmund and Passau (D), DDC (DK), CISE (E), Syseca Logiciel (F), University of Strathclyde (UK)

RAISE: Rigorous Approach to Industrial Software Engineering
KEY WORDS: Vienna Development Method, specification, transformation.
PARTICIPANTS: DDC and BBC Nordisk BB (DK), Standard Telephone & Cables and ICL (UK)

METEOR: An Integrated Approach to Industrial Software Development
KEY WORDS: requirements analysis, specification, ADT, object-oriented paradigms, temporal aspects.
PARTICIPANTS: Philips-RLE and CWI (NL), CGE and LRI (F), COPS (IRL), Philips-RLB (B), Tech. Software Telematica (I), University of Passau (D)

TOOL-USE: An Advanced Support Environment for Method Driven Development and Evolution of Packaged Software
KEY WORDS: methods, application area and target system as parameters of an advanced SEE.
PARTICIPANTS: CISI and CERT-DERI (F), UCL (B), GMD-KA (D), Generics-LTD (IRL)

ASPIS: Application Software Prototype Implementation System
KEY WORDS: reuse of knowledge, (analysis, design and reuse) assistant, knowledge representation formalism.
PARTICIPANTS: OLIVETTI and Tecsiel (I), CAP SOFETI (F), GEC (UK)

GENESIS: A General Environment for Formal Systems Development
KEY WORDS: logic and metalogic, transformation rules, proof rules, tactics, environment generator.
PARTICIPANTS: IMPERIAL COLLEGE and IST (UK), Philips-RLE (NL)

KNOSOS: A Knowledge-Base Environment for Software System Configuration Reusing Components
KEY WORDS: methods, components reuse.

PARTICIPANTS: ESI and Yard (UK), Domier (G), CIT-Alcatel, CNET and MATRA (F)

GIPE: Generation of Interactive Environment
KEY WORDS: semantic description, automatic generation, natural semantics.
PARTICIPANTS: SEMA-METRA and INRIA (F), BSO and CWI (NL)

DRAGON: Distribution and REUSABILITY of Ada-Real-Time Application through Graceful and Online Operations
KEY WORDS: module interfaces, plant configuration management, target system.
PARTICIPANTS: DORNIER (D), TESCSI SOFTWARE (F), University College of Wales and University of Lancaster (UK)

REPLAY: Replay and Evaluation of Software Development Plans Using High-Order META Systems
KEY WORDS: top-down replay, bottom-up assembly, development plan control, modelling of operational properties, expert system for description and generation of assembly plans.
PARTICIPANTS: II and CERT (F), ALPHA (GR), CRI (DK) E2S and UCL (B)

PRACTITIONER: Support System for Pragmatic Reuse of Software Concepts
KEY WORDS: program concepts at design level, reuse.
PARTICIPANTS: PCS and Brown Boveri Cpy (D), CRI A/S (DK)

2.2.2.7 REPLAY

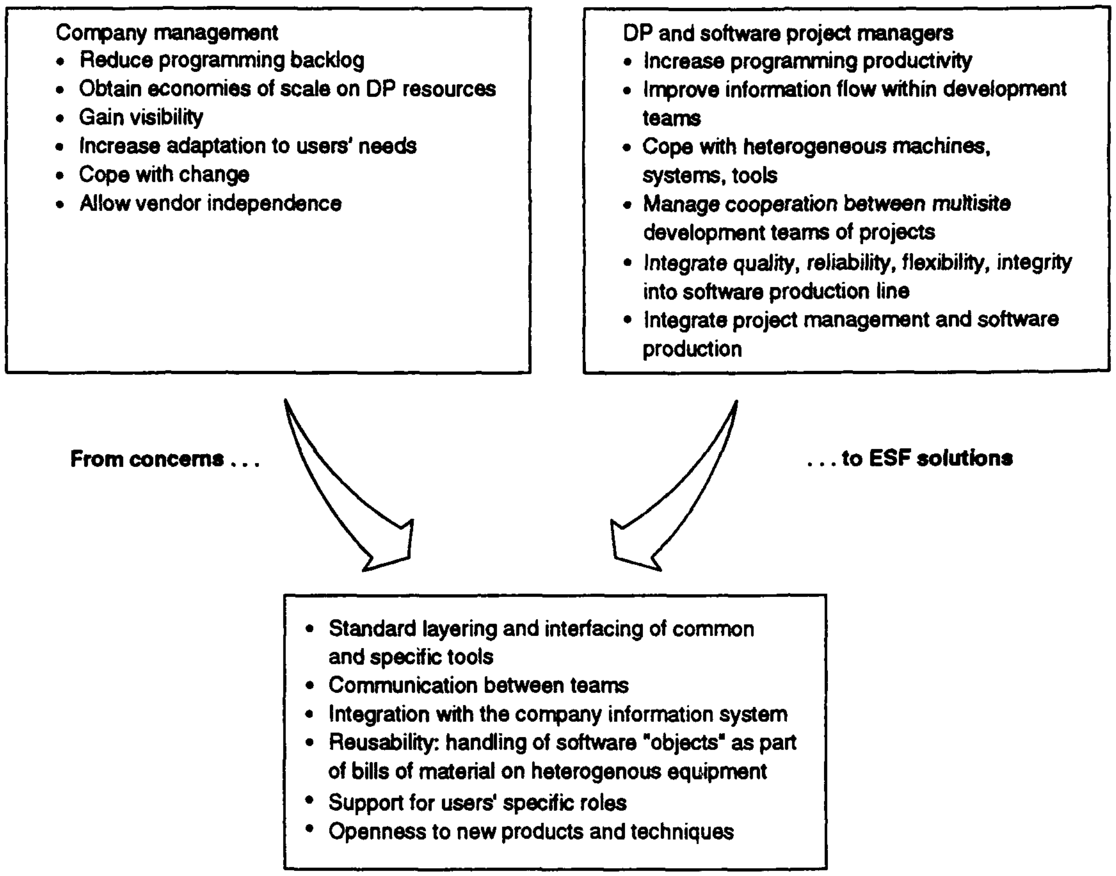
REPLAY is an experimental research project sponsored by ESPRIT. In essence, REPLAY is a meta system that is used to build development plans, store them, and modify them for use in subsequent projects. Another goal of REPLAY is to enhance the reuse of component modules within the development plans. Thus, REPLAY has characteristics of a system to build and maintain both top-down development and bottom-up components.

The following domains are being placed for replay of development plans: proofs, transformation sequences, development of process, development of programs and configuration.

2.2.2.8 Eureka Software Factory

The Eureka Software Factory (ESF) program is a consortium of 13 European companies and research institutions with the goal of improving the efficiency and quality of software production. The motivation for the project is that modem business demands industrially-produced software, where the terms are taken from industry or the factory and applied to software. The ESF approach relies on communication between software modules and software tools, reusability of software components, information-based software engineering and control, ability to cope with a growing variety of methods, and tools on noncompatible machines.

ESF supports the interaction of all team levels, such as managers, developers, and customers, as well as the tasks, responsibilities, interfaces, deliverables and tools (figure 2-11). The ESF uses the concept of *object*. Modules, tools, and software components are considered as objects of a certain type, which encapsulate details for easier manipulation. The object-oriented approach combines openness, integration, and reusability. (See



Company management
- Reduce programming backlog
- Obtain economies of scale on DP resources
- Gain visibility
- Increase adaptation to users' needs
- Cope with change
- Allow vendor independence

DP and software project managers
- Increase programming productivity
- Improve information flow within development teams
- Cope with heterogeneous machines, systems, tools
- Manage cooperation between multisite development teams of projects
- Integrate quality, reliability, flexibility, integrity into software production line
- Integrate project management and software production

From concerns . . .

. . . to ESF solutions

- Standard layering and interfacing of common and specific tools
- Communication between teams
- Integration with the company information system
- Reusability: handling of software "objects" as part of bills of material on heterogenous equipment
- Support for users' specific roles
- Openness to new products and techniques

Figure 2-11. ESF Structure

section 2.2.2.4 for a fuller discussion of object-oriented programming as a paradigm for reuse.) The intent is to open use to all people, roles, computers, operating systems, and languages.

The ESF software engineering reference model provides a predefined structure in which software components are placed and related according to their nature. The architecture consists of several levels, each providing for the exchange of particular kinds of objects. The higher the level, the richer the semantic content and quality of component integration achieved both externally and internally. The user is provided with a uniform view of the software in progress, a unified set of representatives, and expected behavior of the system.

The reference model consists of three levels, which are (top-down), user interaction, tools, and basic services. Interaction protocols, also called software buses, establish the rules that allow communication between adjacent levels.

The potential evolutionary reuse of components, old data types with new tools or old tools with new data types, can be accommodated by the use of standardized modes and levels of communication. Existing components are adjusted or transformed to fit into the architecture.

The architecture is being designed to be both generic and specific: generic to allow multisite communications between heterogeneous equipment, and specific to adapt to an organization's development requirements.

2.2.2.9 Japanese Software Reuse Efforts

During the first phase of the CAMP program, three members of the team visited six Japanese companies to review reusability programs, tools, and techniques used by these companies (ref. 2-32). The application areas review covered a spectrum of activities which included defense systems, office automation, information processing, telecommunications, and consumer and industrial products.

At the time the visits were made and at least within the six companies visited, reusability efforts appeared to be equivalent to those found in U.S. companies. The Japanese efforts have been built around two approaches: system tailoring and using application generators.

The system tailoring approach was observed in most of the companies visited. This process tailors an existing system to meet a new set of requirements. The approach is based on the premise that, within an application domain, most systems tend to have a common structure so that it becomes cost effective to start the development of a new system from a baselined version of a similar system. The baseline can, in turn, be modified to achieve the new set of objectives.

Application generators have been used for several years within very narrow and well-defined application areas, such as banking and man-machine interface construction. The generator system produces a new software system when provided with the requirements of the new application. The high productivity obtained results from the high level of commonality found in banking systems, for example. Application of this technique to embedded systems has not yet been successful. The application of reusable components is being investigated. However, progress has been made toward the implementation of a system.

2.3 APPLICABILITY

2.3.1 Guidance and Control Applications

The modern guidance and control system is becoming a highly integrated structure. Besides the primary flight control function, many other functions (such as integrated flight/propulsion/fire control and trajectory generation and management) are being coupled to enhance mission capability and vehicle performance. Other elements integrated into the system include surface reconfiguration, inertial data and air data sensing, and management of utility systems. This highly coupled and integrated system is characterized by a large amount of sensor interfaces and sensor processing, processing of a multitude of coupled control loops, and processing of output commands to control effectors. Many of these functions involve flight safety and must be performed in a very reliable and uninterruptible manner to prevent loss of the aircraft. To ensure against these potential catastrophic system failures, sophisticated fault tolerance techniques are employed, such as redundant elements, inline monitoring, and reconfiguration.

Included in the software required to implement a typical primary flight control system, are control laws (20% to 30% of total system memory) and failure management (70% to 80%). These values appear to hold in a variety of applications, including both military and commercial systems. The development of control laws for these systems is dependent on the particular vehicle dynamics, taking into consideration both stability and performance requirements. However, the form of the control laws remains similar from application to application. Typical functions within the control laws include schedulers, filters, look-up tables, and limiters. Thus, there is good opportunity for reuse of control law software, particularly in the area of algorithms. By coupling databases with automatic code operators that take into account vehicle dependencies, the potential for large productivity increases in control law development can be achieved. At the higher levels of integration, integrated controls, flight and propulsion coupling, trajectory and energy management, navigation, fire control, Kalman filters, and other related functions have high potential for reuse, again tailored to aircraft-dependent parameters.

Failure management software represents a major portion of a control system and is the most time consuming in development and verification. This type of software holds potential for reuse in many areas, such as input/output signal selection, voting logic, inline monitoring, and functional redundancy through signal synthesis methods.

2.3.2 Implications of This Analysis for the Development Process

The major implication of this analysis for the development process is the need to design software modules and systems with future reuse in mind. This requires using methodologies that support the development of reusable modules. Such a methodology should support the use of standard interfaces and should have tools that aid in developing and supporting systems composed of reusable modules and documentation.

2.3.3 Reusable Software Development System Functional Description

A functional description of a reuseable software development environment is illustrated in figure 2-12. The environment is configured in a manner which gives the G&C engineer the ability to prototype and develop a system from a library of reusable software objects, ideally through all cycle phases. In its elementary form, the system permits the user to select objects from a library, assemble the objects into a system, perform appropriate tests, and analyze the results to determine conformance with requirements. The input to the system is system requirements with the output being code and documentation. The system interface should be flexible to permit use by a broad range of users and contain an assortment of interface techniques such as: graphic, natural, or formal languages. The system must have the ability to enforce the use of existing objects, when they meet requirements, and the flexibility to accommodate newly developed objects.

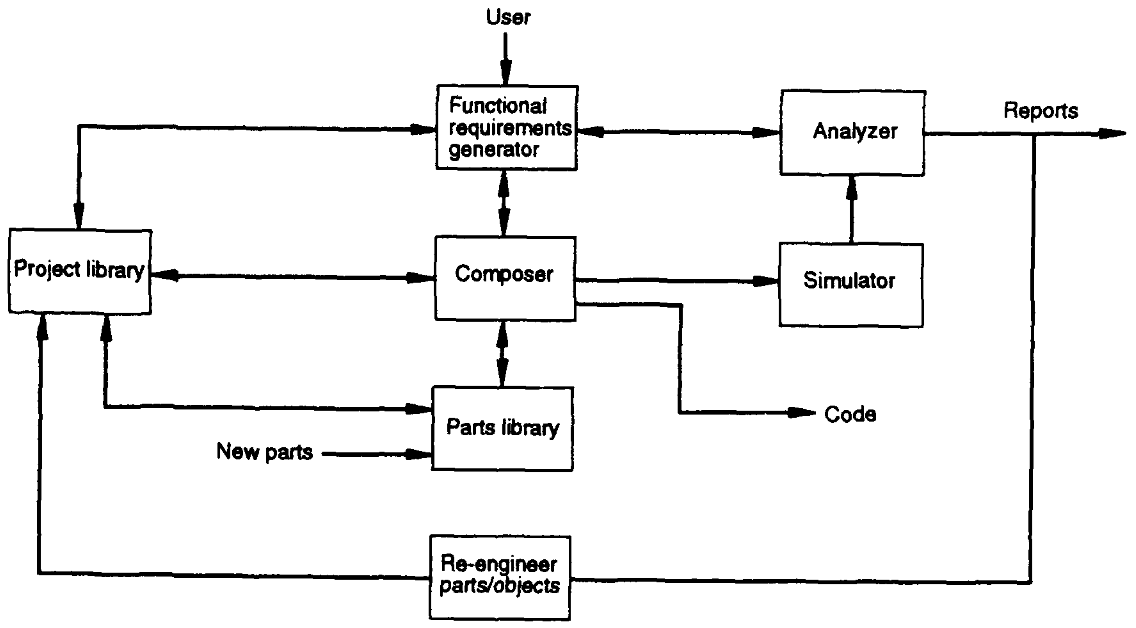


Figure 2-12. A Reusable Software Development Environment

The main interface into the system is through the functional requirements generator. This part of the system allows the user to build a software specification and OFP descriptors from the system requirements. The information generated is utilized by the composition part of the system to query the parts library to determine if a usable object exists for direct use, if a new element must be developed, or if an existing element must be re-engineered for the new system. The generator also provides a link into the analyzer to determine how well the software meets system performance requirements. Information generated is stored into the project library to form a database specific to the system under development.

The parts library contains software objects such as those described earlier in table 2-4. Standard descriptors are employed to characterize the functionality of elements such as filters, redundancy management, Kalman filters, etc. Other attributes such as: I/O, timing, size, and testing conditions are also provided. The library must also provide a mechanism for cataloging and managing the components contained within the library and other components imported into the library from other projects or from a system currently being developed.

The project library contains information unique to the system under development. This information can consist of software components, testing information, specifications, software metrics, and other project management data. New software can be, under the current development, exported to the parts library when appropriate.

The composer portion of the system orders suitable components from the parts library and integrates these elements into an appropriate configuration to meet the software specification requirements. Code generators are also provided to produce code which can be executed by the simulator and code which can be loaded into system level simulators and target computers. The composer also aids in the re-engineering to existing components or the development of new components to meet project specific requirements. Another function of this portion of the system is to generate appropriate test generators to evaluate system performance compliance.

The simulator allows testing at all levels: module, open loop, and closed loop. Appropriate aircraft, sensor, and effector models can be imported from the parts library to build an effective system level simulation.

The analyzer receives system performance requirements and with the data produced by the simulator analyzes results to determine requirements conformance and thus validates the system design. The simulator produces appropriate reports and also issues problem reports which indicates where re-engineering of certain portions of the developed system are in order.

This description of a reusable software development system is based strictly on the reuse method for enhancing software productivity. It is obvious that by combining this methodology with the other methods described later in section 6 of this report, that a greater synergism is achievable with a higher payoff potential.

2.4 LIMITATIONS

2.4.1 Libraries

As the demand for cost-effective software increases, reuse becomes more important as a potential solution to low programmer productivity. To investigate this potential solution and promote reuse, a number of groups have developed reusable software libraries. Most of these libraries cater to Ada which supports software reuse through package and other generic features. However, many of the reusable software libraries accept and support reusable software written in other languages.

Intermetrics's reusable software library (RSL) software classification scheme and its database of software attributes have been designed to simplify the selection of reusable components. Furthermore, Intermetric's RSL tools help software developers find and evaluate components that meet their requirements. These library tools have been integrated with software design tools, making reuse a natural extension of the design process. Additional tools automate the work of the librarian, who must enter the component attributes into the database and maintain the RSL.

Intermetrics defined a long-term phased development plan to produce a library suitable for large software development projects. Rather than define an elaborate reuse library to be implemented in a single step, they plan to prototype the system in parts so that they can evaluate the feasibility, utility, and potential of the design as the prototype is developed.

Table 2-7 is an overview of the seven phases in their approach. The phases progress from a small prototype library system to a fully functional, geographically distributed system. First, they focused on developing a software catalog with tools for component evaluation and reuse-oriented design. Three phases are complete and the fourth is underway.

Table 2-7. RSL Phased Development Plan

Phase	Activity
1. Analysis and requirements	• Identify characteristics of previous software libraries • Identify classification scheme
2. Develop initial reusable software library	• Design and implement prototype library • Catalog initial holdings • Provide capability for component evaluation
3. Develop initial reuse-oriented design tool	• Develop interface to the library • Provide support for object-oriented design tool • Support inclusion of reuse components into top-level design • Support automatic generation of Ada PDL
4. Integrate subsystems of the reusable software library	• Develop uniform user interface • Migrate all subsystems of the reusable software library to a common window-oriented workstation
5. Add additional functionality to the reusable software library	• Add tools to support component entry
6. Distribute reusable software library	• Provide the capability of assessing the database from multiple machines across a single company • Provide support for multiple databases within a single company
7. Formulate reusable software library for use by multiple companies	• Develop a reusable software library which can be tailored for the needs of many companies

The library management subsystem provides a set of tools to help the RSL librarian and quality assurance personnel populate and maintain the software library. The functions of the library management subsystem are to extract reuse information from design or source code files, to ensure the quality of the candidate components for the RSL database, and to maintain the RSL. The library management subsystem includes tools to ensure efficient operation. The subsystem features automated data collection, standardized data entry, continuity and consistency of reuse information across the life cycle, completeness and reasonableness of reuse information, and automated examination of reuse information.

The RSL is hosted on the VAX11/750 and can automatically scan PDL and source code files and extract specially labeled reuse comment statements. With this extraction program, the attributes of a candidate component are automatically retrieved from PDL and source code files. The data can then be reviewed for possible entry into the RSL database. The list of labels and corresponding attributes is presented in table 2-8.

Table 2-8. RSL Labels and Attributes

Label	Description
UNITNAME	Name of the procedure, package, or subroutine
CATEGORY CODE	A predefined code that describes the functionality of the component
MACHINE	The computer on which the component was programmed
COMPILER	Compiler used during development of the component
KEY WORDS	Programmer-defined words that describe the functionality of the component
AUTHOR	
DATE CREATED	
LAST UPDATE	
VERSION	
REQUIREMENTS	Information about any special requirements of the component (e.g., other components that must be available)
OVERVIEW	A brief textual description of the component
ERRORS	Information about any error handling or exceptions raised in the component
ALGORITHM	Description of the algorithm
DOCUMENTATION AND TESTING	A description of available documentation about the component and a description of test cases

Selecting the software attributes to extract is really part of a larger question: What information should be retrieved about a particular entry? The list of elements seems to be highly influenced by the size of the library (number of software units stored) and the degree of cooperation among the users of the library.

The RSL software classification strategy is based on the combination of two alternate mechanisms. The first mechanism is the assignment of a hierarchial category code to each component added to the library. A category code specifies the type of the component and its relationship to other components. Sample categories include common math functions, data structures, and sort and search routines. This schemes concept is similar to the one used by computing reviews and the IMSL library.

The second mechanism permits up to five descriptive key words for each component. These key words are not associated with the category codes allowed for overlapping topics (because packages do not always conveniently fall into strict tree classifications), and can grow with the project's needs (without reprogramming and without an all-knowing database administrator).

Central to the evaluation process (called Score) is the identification of software application domains, because application domains determine which functional and qualitative attributes are most significant when examining components for reuse. Functional attributes describe what the software component does and how it is implemented. Qualitative attributes provide objective and subjective metrics and rate the quality of software components. Objective metrics include line counts and complexity measurements. Subjective metrics include readability, program structure, programming style, documentation, and testing. For Score to accurately evaluate components against the users of software requirements, all components must be rated against the same yardsticks. Standard metrics must be established for each qualitative attribute, and guidelines must be defined for uniform grading practices. Consistent quality ratings are especially important for the subjective metrics.

The user invoking Score is prompted to specify the required application domain and to indicate the importance of each attribute that Score considers when evaluating components in this domain. Based on this information, Score searches the library for candidate components, evaluates them against the user's requirements, and rates them according to a scoring algorithm.

A similar method for evaluating components was developed by Prieto-Diaz and Freeman. Their approach uses fuzzy logic, with user experience serving as a modifier to evaluate similar components. The Intermetric approach differs in that it lets users explicitly describe the relative importance of software attributes.

Figure 2-13 illustrates a user's abridged session with Score. After the user is asked some preliminary questions to determine what type of component is needed, the Score barometers are displayed along the screen. A mouse is used to click on the arrows beneath each barometer level to reflect the relative importance of the software attributes to the application.

Request	Response	Action
I need a stak package.	by "stak" do you mean "stack"? Current series of requests: I need a stack package.	The unit names and overviews of all stack packages are displayed.
Only display those that implement garbage collection.	Current series of requests: I need a stack package. Only display those that implement garbage collection.	The above search is pruned so that only those stacks that are managed are displayed.
When were they written, and what version are they?	Current series of requests: I need a stack package. Only display those that implement garbage collection. When were they written, and what version are they?	In addition to unit names and overviews, the dates and version numbers of the packages from the previous search are displayed.

Figure 2-13. Software Attributes Extracted from Reuseable Components

In the second Score screen, the user indicates whether the component's operations, programming language, level of testing, quality of overview and external documentation, and set type (for example, multiset) are especially important.

Score then generates additional questions based on the barometer levels. After these questions are answered, a list of components is presented in the order of their suitability. The user may now choose to end the session or to reset the barometers. Resetting the barometers will result in a reordering of applicable components. After exiting the Score system, the user is returned to the RSL main menu, and can generate a report of the ordered components.

2.4.2 Business Practices and Liabilities

It appears that the biggest payoffs from reusability are those derived during the design, implementation, and integration phases of a software life cycle. There are also large payoffs in the maintenance phase of the life cycle because of reduced need for testing when a reusable module is added to an existing system; the module has been used and tested in previous applications. In addition, the maintenance costs are reduced because reusable modules inherently have characteristics that enhance maintainability, such as well-specified interfaces, cohesiveness, good documentation, and so on. Because there are large payoffs, reusability will become a major software generation tool of the future. However, many costs are encountered when a program of reusable software is set up. These costs include the following:

a. Need for more software system analysts and designers and fewer programmers.

b. Reusable objects will have to be identified, analyzed, and integrated with other objects. Therefore, more systems analysts may be required and more (proportional) time may be spent at the front end of a software development cycle. This may cause some managers to become concerned, especially those used to following traditional life cycle development models.

c. Building reusable software costs more the first time. The program manager who faces cost and schedule restrictions will not be inclined to develop standard, reusable software. The program manager knows that it will cost more and, without an incentive of some kind, may elect to develop software that is not necessarily reusable.

d. It costs to maintain libraries of reusable software objects. Resources, both in personnel and facilities, are required to maintain the software library. Automated search and retrieval tools are required as well, and there are costs to develop and service these tools.

e. Users of the library must be trained. Procedures for using the library and guidelines for reusing software are required.

Faced with these costs, many organizations will not want to expand the software reuse concept. More return-on-investment information is needed and incentives from military customers to industry may be necessary before software reuse is widely accepted.

Legal issues must be faced as well. Currently, ownership and liability of reusable software remains an issue not well developed in the legal system. Software reused within an organization does not pose a legal problem, but software transferred from one organization or company to another creates legal problems of ownership and liability. Liability issues are also a problem, not only where performance expectations and safety are involved, but also for the one who is responsible for maintenance.

Software reuse will be hindered by these legal issues, but the business advantage will keep pressure on the legal system to solve these issues. Also, the increase in sales of software and use of shareware products for personal computers and Macintoshes may help resolve some of the legal issues.

2.4.3 Standards

There are basically three types of standards to consider: (1) standards for storing and retrieving modules and information about modules (such as database or library standards discussed in a previous sections of this chapter), (2) the syntactic interface (i.e., the protocols for handling input and output parameters and for invoking the module) and (3) the semantic interface (i.e., the module's functional performance). The last two are considered to be the specification of the syntactic and semantic module interfaces.

The specification level of the syntactic and the semantic interfaces will depend on the methodology or system of reuse. Those systems that are language dependent will tend to rely on the implementation language (or a PDL) to define the interfaces. Many languages, such as Ada, have sufficient mechanisms to specify precisely the syntactic interface, but no language has sufficient mechanisms to specify the semantic interface precisely. This is normally done by describing the module's function and performance in English, which generally leads to an imprecise specification. Other standards that are used may resort to procedural specifications or specifications of changes to data structures. In many cases, the procedural specifications of an interface are as complex as the module itself, giving little insight that could not be gained by inference from the module's source language. Such attempts at procedural specification often lead to the idea that a translator should be produced to convert the specification directly into code. The result is a slightly different programming language, not a useful abstract specification language.

Following is a set of standards that should be followed in the specification of abstract interfaces for reusable software modules. The interface—

a. Should not disclose any of the implementation details of the module.
b. Must present a concise description of the facilities available from the module, in terms of effects that are observable to the user.
c. Should be divided into sections and formatted so that a reader unfamiliar with the module is able to find a piece of information without having to study the entire interface specification; it should serve the quick-reference reader as well as the first-time reader.
d. Should not provide unnecessary duplication of information, which would make using and maintaining the specification more difficult.

A standard interface specification method that has these properties is described in reference 2-20. It is a language-independent specification method which can be used as the basis for a standard interface description in a system designed for software reuse. The following is a suggested standard interface specification based on reference 2-20 and modifications in reference 2-33. The specification should consist of four sections. The *Introduction* section should contain a brief prose description of the module; the *Syntax* section should contain a description of the syntax of the interface; the *Effects* section should describe the normal behavior, or semantics, of the module; and the *Exceptions* section should contain a description of the module semantics when it is called with illegal parameters (or out-of-bounds parameters). As with any standard, it is desirable to have the specification as precise and formal as possible; however, using formalisms tends to make the specification obtuse and difficult to use in practice. This is true for any of the currently usable formal specification languages. Thus, the designer is faced with a choice between being as formal as possible and keeping the specification understandable to the users. Hoffman, in reference 2-33, suggests the following practical guidelines on choosing between prose (or structured prose) and formal specifications.

Consider some feature f to be specified. f should be specified in prose if:

a. f cannot be specified formally.
b. f can be specified formally, but only in a way that will take more time than can be justified. A policy of "formality at any cost" simply does not make sense economically.

c. It is known how to specify *f* formally, but not in a way that is readable by the user. A completely formal specification is of little value if it cannot be read.

Hoffman adapted the abstract interface concepts in reference 2-20 and implemented them using C, Pascal, and Z80 assembler language. He also translated the interface specifications into Trace, a formal specification language (ref. 2-34).

2.5 FUTURE RESEARCH AND DEVELOPMENT

No great breakthroughs in technology are needed to enhance software reusability. The use of existing technologies and methods could substantially increase software reuse. The following sections describe techniques that are crucial to enhancing software reusability.

2.5.1 Software Development Methodologies

The most crucial issue in the reuse of a software module is that the module must have been designed with reuse in mind. Therefore, software reuse will not be commonplace until methodologies are used that support the initial development of reusable modules. This does not imply the need for future research and development in software methodology, but it does imply the need to adopt existing methodologies that are designed to support the development of reusable modules (see, for example, reference 2-21). Such methodologies should support standard interfaces, should have tools that aid in the development of reusable modules and supporting documentation, and should have tools that support the design and maintenance of systems that use reusable software modules. These ideas are discussed in the following sections.

2.5.2 Standard Interfaces

The use of standard interfaces and module descriptions is a requirement for software module reuse. Many existing interface description models serve as a basis, but, as described in section 2.4.3, no one system is completely satisfactory for the specification of a module's syntactic and semantic interface. The adoption of a standard software module interface for projects is clearly necessary. Such an interface standard must have the proper tradeoff between formal and informal specification methods (see, for example, reference 2-33), and it should have the characteristics described in section 2.4.3.

It appears that a relatively small amount of research and development effort is needed to develop such a standard system given the advances that have been made in the last decade. The larger effort will be to enforce its use.

2.5.3 Support Tools

For the most part, existing software development tools are useful in software reusability. The only specific new tools needed are those related to maintaining and accessing libraries of reusable objects. Standard database management systems can be modified for use in libraries of reusable objects.

2.5.4 New Cost Models

Over the past two decades, various models have been developed to give program managers, systems analysts, and technical team leaders the means for estimating software parameters such as resources used, productivity, and quality. These software measures are used to quantify how well we are meeting the goal of software engineering: the production of higher quality software at lower cost. Before discussing the requirements for new or updated models, it will be advantageous to review current techniques to assess practices and identify deficiencies.

Many metrics have been proposed to measure software development productivity or, alternatively, production costs (refs. 2-3, 2-35, and 2-36). Although lacking precise definition, lines of code per man-month has remained the most commonly used metric even though its interpretation remains varied. However, it is an aesthetically pleasing parameter since it can be easily linked to typical measures of productivity for a physical system (for example, the number of cars produced per month). Some of the major models currently used to predict cost are listed below.

a. Aerospace Corporation.
b. Boeing Computer Services.
c. Constructive Cost Model (COCOMO).
d. DOD Microestimating Procedure.
e. Doty Associates, Inc.
f. ESD - SPSCEM.
g. Farr and Zagorski.
h. Hahn and Stone.
i. IBM Walston-Felix.
j. Price-S (RCA).
k. SLIM (Putman).
l. Tecolote Research Corp.
m. TRW.
n. Wolverton.

Many of the models used to predict development cost and time of software projects have been developed by applying empirical relationships for cost per statement and time to develop a line of code. While this approach has met with success on relatively small projects, it begins to become ineffective as the project and resulting complexity become large. It is difficult to arrive at a good estimate of the lines of code, because they become more intuitive. Furthermore, misinterpretations can arise from use of data from different organizations or data from the same organization, but gathered under a different set of rules. The models also require recalibration when applied in different development environments or application domains, such as business versus embedded systems. Many of the predictive models depend on estimates of variables, such as program size. Since this parameter is difficult to estimate at the early development stages, the resulting prediction of resources becomes inaccurate. This is especially true for large, complex, new applications.

To enable the development or refinement of more accurate cost models, several advancements must be pursued. Metrics parameters and other appropriate measures definitions must be standardized so that they can be applied uniformly across the software development industry. This will aid in the development of automatic and unobtrusive data collection tools that are embedded in the development environment. It will also provide the capability for objective measurements of productivity and cost as opposed to subjective judgments now used in some models. Present models are based on the waterfall development cycle; as other cycles come into use, models must be refined or newly developed. To judge the application of reusable software for a given system, new models that quantify the payoffs of reuse over the system life cycle must be developed and verified. In all cases, verification of the models are required over many systems, application domains, and development environments.

2.6 TECHNOLOGY TRANSFER

Technology transfer is a problem in almost all aspects of software technology. There are many reasons for this, but two of the major reasons are: (1) some technology is transitioned before it is actually ready, making system managers wary of all technology; and (2) many software managers and workers are reluctant to accept any new technology that significantly changes the way they do business. The transfer of technology that would enhance software reuse is affected by these two general problems and, in addition, by problems unique to the particular technology. These problems include the high initial startup cost, the lack of standard interface description techniques, the *not-invented-here* syndrome, the actual or perceived need for efficiency, the lack of support tools, the business implications of widespread reuse, and the dearth of existing software modules that can be reused.

If an organization is to succeed in software reuse, it must develop the software in each project so that it can be reused in other projects. This increases the cost of developing the software for the particular project, and, in the initial stages of implementing the reuse technology, there is no payoff in terms of software that can be reused. Thus, the startup costs are high for the projects that are in development when reuse technology is initiated. To succeed, the company or organization must have a global view of the savings from the reuse technology.

The lack of standard interface description techniques used within an organization inhibits the use of modules designed by separate projects within the organization. (The lack of interorganizational standards, of course, inhibits the use of software modules among different organizations.) The characteristics needed for a standard software interface description are given in section 2.4.3.

The *not-invented-here* syndrome has a number of manifestations. In many cases, the designers of the current system do not trust software written by other people. In some cases, the existing software performs a task that is similar to the required task, but the cost of modifying it is too high. The software specifications may be nonexistent or too ambiguous to allow the designers of the new system to have any confidence in the performance of the module. The actual or perceived need for efficiency leads to the production of custom software modules; these modules may be written in languages that are not conducive to subsequent reuse. In some cases, the methodologies that would support the development of reusable modules do not have an adequate set of tools to support the design, storage, and retrieval of reusable software modules. In many cases, the software methodologies that are supported by adequate toolsets are not suitable for producing reusable software modules.

Many companies are concerned about questions of ownership, maintenance, and liability related to software modules.

Finally, the lack of an existing pool of reusable software modules discourages the transfer of technology that is needed for the reuse.

The other sections of this chapter have clearly indicated that viable technology exists to support software reuse and that cost savings can be substantial. So, the question is how this technology can be transitioned into an organization. The most successful paradigm for transitioning technology is to provide the technology and a demonstration of the technology on a system that is relevant to the organization receiving the technology. This has been successfully employed in the Common Ada Missile Program (ref. 2-32) and the Software Cost Reduction Project (ref. 2-19). It is the approach being used in STARS (see section 2.2.1.2 describing the STARS reusability effort) and by the Software Productivity Consortium (ref. 2-21).

This approach appears to be reasonable for transferring the technology into common use within its software development cycle. To pursue this approach, the first step is to define the interface standards and methodogy used in support of reusability. Then the tools needed to support the process are acquired, including the library support tools, and an integrated package provided for use by the software developers. After the methodogy and tools are provided, they are tested on a typical software development project. The results are used as models for the software designers and developers to follow in the routine development of software for the organization.

2.7 REFERENCES

2-1 Reference Manual for the Ada Programming Language, U.S. Department of Defense, Washington, D.C.

2-2 Reusability Guidebook, version 5.0.

2-3 A Guidebook for Writing Reusable Source Code in Ada, Honeywell Computer Science Center Technical Report, Honeywell Computer Sciences Center, Golden Valley, MN, March 1986.

2-4 Booch, G.; Software Components With Ada, The Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1987.

2-5 Parnas, D.; "Designing Software for Ease of Extension and Contraction," Proceedings of the 3rd International Conference on Software Engineering, 10-12 May 1978, pp. 264-277.

2-6 Parnas, D.; "On the Criteria To Be Used in Decomposing Systems into Modules," Comm. ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058.

2-7 Hester, S.D., Parnas, D.L., and Utter, D.F.; "Using Documentation as a Software Design Medium," The Bell System Technical Journal, Vol. 60, No. 8, October 1981, pp. 1941-1977.

2-8 Heninger, K., Kallander, J., Parnas, D., and Shore, J.; Software Requirements for the A-7E Aircraft, NRL Memorandum Report 3876, 27 November 1978.

2-9 Heninger, K.; "Specifying Software Requirements for Complex Systems: New Techniques and their Application", IEEE Transactions Software Engineering, Vol. SE-6, pp. 2-13, January 1980.

2-10 Clements, P. and Parnas, D.; "A Rational Design Process: How and Why to Fake It", IEEE Transactions on Software Engineering, Vol. SE-12, Number 2, pp. 251-257, February 1986.

2-11 Parnas, D. and Weiss, D.; "Active Design Reviews: Principles and Practices", Proceedings, Eighth International Conference On Software Engineering, pp. 132-136, 1985; also available as NRL Report 8927, 18 November 1985.

2-12 Parnas, D., Clements, P., and Weiss, D.; "The Modular Structure of Complex Systems", Proceedings, Seventh International Conference on Software Engineering, pp. 408-417, March 1984; reprinted in IEEE Transactions on Software Engineering, Vol. SE-11, pp. 259-266, March 1985.

2-13 Britton, K. and Parnas, D.; A-7E Software Module Guide, NRL Memorandum Report 4702, 8 December 1981.

2-14 Guttag, J. J. and J. V. Horning; "The Specification of Abstract Data Types", Acta Informatica, 10, 1978.

2-15 Clements, P., Parker, A., Parnas, D., Shore, J., and Britton, K.; A Standard Organization for Specifying Abstract Interfaces, NRL Report 8815, 14 June 1984.

2-16 Clements, P., Faulk, S., and Parnas, D.; Interface Specifications for the SCR (A-7E) Application Data Types Module, NRL Report 8734, 23 August 1983.

2-17 Clements, P., Parnas, D., and Weiss, D.; "Enhancing Reusability with Information Hiding," Proceedings of a Workshop on Reusability in Programming, pp. 240-247, September 1983.

2-18 Parnas, D.L.; "A Generalized Control Structure and its Formal Definition", Communications of the ACM" ,Vol. 26, No. 2, pp. 115-123, August 1983.

2-19 Faulk, S. and Parnas, D.; "On Synchronization in Hard-Real-Time Systems", Communications of the ACM, Vol. 31, No. 3, March 1988.

2-20 Clements, P., Parker, A., Parnas, D.L., Shore, J., and Britton, K.; A Standard Organization for Specifying Abstract Interfaces, NRL Report 8815, 14 June 1984.

2-21 Weiss, David M.; Reuse and Prototyping: A Methodology, SPC-TR-88-022, Version 1.0, March 1988.

2-22 Parnas, D.; "On the Design and Development of Program Families," IEEE Trans. on Software Engineering, Vol. SE-2, No.1, pp. 1-9, 1976.

2-23 Parnas, D.L.; Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems, NRL Report 8-47, 3 June 1977.

2-24 Basili, V., Caldiera, G.; Reusing Existing Software, University of Maryland Computer Science Technical Report UMIACS-TR-88-72, October 1988.

2-25 Basili, V., Rombach, H.; Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment." University of Maryland Computer Science Technical Report UMIACS-TR-88-92, December 1988.

2-26 Meyer, B.; " Reusability: The Case for Object-Oriented Design," IEEE Software, March 1989.

2-27 Bobrow, D. G., and Sefik, M. J.; Loops: An Object-Oriented Programming System for Interlisp, Xerox PARC, Palo Alto, CA, 1982.

2-28 Cannon, I. J.; Flavors, MIT Artificial Intelligence Lab, Cambridge, MA, 1980.

2-29 Goldberg, A. and Robson, D.; Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.

2-30 Stroustrup, B.; The C++ Programming Language, Addison-Wesley, Menlo Park, CA, 1986.

2-31 Meyer, B.; "Eiffel: Programming for Reusability and Extendability," SIGPlan Notices, 1987.

2-32 McNichol, D.G., Palmer, C., et al.; Common Ada Missile Packages, Vol. I and Vol. II, Technical Report AFATL-TR-85-93, May 1986.

2-33 Hoffman, D.; "Practical Interface Specification", Software–Practice and Experience, Vol. 19(2), February 1989.

2-34 McLean, J.; "A Formal Method for the Abstract Specification of Software," Journal of the ACM, 31, 1984.

2-35 Conte, S., Dunsmore, H., and Shen, V.; Software Engineering - Metrics and Models, The Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1986.

2-36 Putnam, L.W. and Wolverton, R.W.; Tutorial, Quantitative Management: Software Cost Estimating, Computer Software and Applications Conference, November 1977.

2.8 BIBLIOGRAPHY

[DIM] Britton, K., Parker, A., and Parnas, D.; "A Procedure for Designing Abstract Interfaces for Device Interface Modules", Proceedings, Fifth International Conference on Software Engineering, pp. 195-204, 1981.

[GRACE] Margono, J. and Berand, E.V.; "A Modified Booch's Taxonomy for Ada Generic Data-Structure Components and Their Implementation."

CHAPTER 3

KNOWLEDGE-BASED APPROACH TO SOFTWARE GENERATION

3.1 OVERVIEW

In this chapter some reflections are presented about using artificial intelligence (AI) especially expert systems, for "automatic" software generation. The main lines of this approach are:

a. Guidance and methodological support of activities related to the software development process.

b. Automating the software development process.

c. Expert systems as part of the software development life cycle.

The state of the art of this approach is outlined in section 3.2. In section 3.3, the support of software engineering methods is presented, particularly the application of AI methods to establish active expert assistance during the software development process.

In section 3.4, ways to automatically generate software systems are discussed. Several approaches have been made and there have been some promising advances for commercial application. But, unless it is possible to effectively utilize the knowledge of all persons involved in creating guidance and control systems (users, system experts, system designers, and software engineers) there will be no real progress in computer-aided support for building these systems. Applying knowledge-based approaches to software development is still an area of research. The approaches presented here are based on the experience of the companies MBB and TST of the Deutsche Aerospace.

Section 3.5 describes associated problem areas that are of particular interest within the framework of guidance and control systems.

To achieve knowledge-based approaches to software development, much research and development is necessary. The most important aspects that must be investigated and improved in the future are outlined in section 3.6.

3.2 CURRENT STATUS

Software development environments of the second generation are presently being established for the development of embedded computing system (ECS) mission- and safety-critical software (such as guidance and control systems) in defined large projects, such as EFA or the Columbus space station. However, for these environments, expert systems could only provide some overall user guidance, if incorporated at all.

Expert systems providing comprehensive and effective support as well as AI approaches to software development are being studied. Promising solutions are being proposed to use these expert systems in third generation ECS development and project support environments in the mid-1990s.

There are many American and European initiatives and projects contributing to this goal. Some of the most promising are:

a. Software Technology for Adaptable Reliable Systems (STARS). This program is sponsored by the U. S. Department of Defense (DoD) and is budgeted at $250 million. STARS is aiming at the stimulation of research and development in the software technology field to improve the productivity of software development environments.

b. Software Productivity Consortium (SPC). SPC is a joint 5-year effort of the U.S. aerospace and defense industry to develop a tool set that will increase software productivity. It is budgeted at $45 million and employs 170 experts.

c. Defense Advanced Research and Project Agency (DARPA) Strategic Computing Initiative (SCI). The heavily-funded SCI is trying to push forward the state of the art in AI and introduce it in military systems. Some of these programs deal with the enhancement of software development.

d. Aerospace Intelligent Management and Development Tool for Embedded Systems (AIMS). AIMS is a joint European effort by the five leading aerospace companies, AIT, AS, BAE, CASA, and MBB, to improve productivity in the development of embedded computer systems by an order of magnitude and to set a common European standard for future aerospace projects. It is budgeted at 70 MECU over the next 5 years.

The first project using a knowledge-based approach to improve the programmer's skills was the "Programmer's Apprentice" undertaken by Massachusetts Institute of Technology (MIT). This project was really the baseline for all AI systems (expert systems, natural language systems, and computer-based training) supporting particular roles relevant to the development of software. However, no breakthrough in this area has been achieved yet.

Also, several attempts have been made to use an expert system approach for specification transformation systems and automated program generation systems. A survey of different approaches is presented in reference 3-1. However, the main area covered is non real-time systems or, at least, nontime-critical systems.

An approach orthogonal to the one described above (which may be labelled the "supportive approach") is based on reengineering a similar product. This reengineering approach changes the existing product, step by step, until it fulfills given requirements. This approach is complicated and only one attempt has been made to support it with a set of tools. As these tools are still in beta testing, any increase in productivity and quality has not been measured.

The current status of research in the expert system approach to software generation is reflected in a collection of references added to the general reference list.

3.3 APPLICABILITY IN SUPPORT OF ENGINEERING METHODS

One of the main goals of an expert systems approach to developing future guidance and control (G&C) systems is to increase productivity and quality. This can be done by:

a. Raising the expertise of most of the project members.

b. Increasing the reusable objects in a project.

c. Using formal specification and new fourth generation language methods to produce better quality and fewer errors.

d. Automating the software generation process.

3.3.1 Expert Assistance

3.3.1.1 Expert Assistance Goals

In most project teams, there is no common level of expertise, that is, some members are much more productive than others. Under higher expertise:

a. More and better (more accurate) information (including good and bad solutions) is instantaneously available, so that better decisions can be made.

b. A better judgement can be made of the impact of decisions because of better analytical skills and wider perspective.

c. Priorities can be identified in areas of crucial concern.

d. Areas can be looked at from different viewpoints and can be rationalized and can be looked at described in other forms. Using heuristic skills in the rationalization process usually leads to recognition of deadends in design or development and of reusable items, thus, improving productivity even further.

Experts not only produce better quality objects, they also produce these objects faster and more accurately according to previous specifications. Expert assistance can be regarded as a computerized expert associate available to all project members. A computerized expert associate could help to solve or examine other classic software engineering problems, as follows:

a. Knowledge in the domain application. Early in the life cycle it is important to have a good view of the application nature and its constaints. Expert assistant might explain the most interesting application characteristics allowing the user to dominate the context better and faster.

b. Supervision of programming standards. Through the use of an expert assistant, a higher uniformity of software objects to a standard programming style may be achieved.

c. Low level of automation. Some simple operations may be performed automatically by the expert assistant.

d. Difficulty in re-using existing software objects. An expert assistant might search a library to find objects with characteristics that match some requirements.

e. Support of maintenance activities. An expert assistant to keep the knowledge about a system could have a better control of changes (ref. 3-2).

f. Cost implications of errors. Through expert supervision, several methodical errors might be avoided or immediately detected.

We can distinguish between a passive and active expert assistance. With the former, assistance is made available on user request; some examples are online help information and documentation. In the latter case, the system has initiative. It may interrupt the user to provide advice (user guidance), or even take partial control of a tutoring session, if the user needs training (computer-based tutoring). Figure 3-1 shows this classification.

Building an expert assistant presupposes dealing with very hard AI problems, such as acquisition of a wide quantity of knowledge about several domains and representation and processing of this knowledge. As a consequence, in addition to classic advice-giving topics, the following chapters will also briefly overview some knowledge engineering issues.

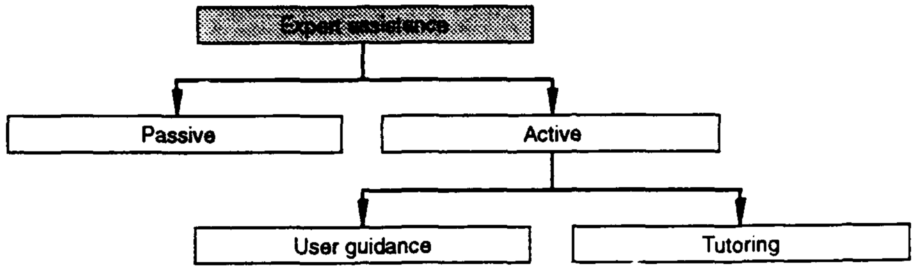


Figure 3-1. The Classification of Expert Assistance

3.3.1.2 Principles and Characteristics

An ideal intelligent assistant should support a user in his activity in an effective, clear, and concise way without frustrating him through excessive or useless interruptions. Support should be provided at different levels, depending on three orthogonal aspects:

a. Advisor scope: range of supportable activities.

b. System initiative: passive or active assistance.

c. Intensity of support: from basic functions to tutoring mode.

To this end the assistant needs several functionalities, ranging from classic online help facilities (ref. 3-3) (documentation, warnings, error notification, etc.) to complex knowledge-based functionalities (monitoring and tutoring strategies, user performance evaluation, etc.). These functionalities may be classified as shown in figure 3-2. There is a class of basic functionalities which should be available no matter what kind of initiative the assistant has. This class gathers all those functionalities that the user may directly invoke when in "passive mode", but which may also be invoked by the assistant when giving some "active" advice. When performing user guidance, the assistant needs more complex knowledge-based functionalities than the basic ones (a user monitoring strategy and criteria for evaluating user performance) in order to provide monitoring and to adapt the support level to the user's skill. If a training activity has to be performed, the assistant will also require one or more tutorial strategies based on some education/training methods.

Tutoring functionalities	Guidance functionalities	Basic functionalities
• Tutorial strategies	• Monitoring strategies	• Online documentation
• Pedagogical knowledge	• User performance evaluation	• Explanation facility
		• Error notification
		• Automatic warnings
		• Heuristics
		• Advices
		Information about:
		• Methods
		• Standards
		• Tools

Figure 3-2. Basic Functionalities

Several of these functionalities require that the assistant has a model or description of the user. The universe of user models can be seen as a three-dimensional space, where the dimensions are:

a. A model of a single canonical user (normative model) or a collection of models of single users (individual differences).

b. Models explicity specified either by system designers or by users themselves; or models inferred by the system on the basis of user behavior (behavior models).

c. Models of long-term user characteristics (such as, expertise or interest areas); or models of relatively short-term user characteristics (for instance, the problem the user is trying to solve).

Each class of models brings benefits and drawbacks. Systems with a normative model can have the model permanently stored inside them, while individual models have to be built at the time. Behavioral models must handle the hard problem of correctness of inferred information; explicit models can avoid such problems, but to the detriment of flexibility. Systems handling short-term knowledge have to notice when the situation changes, while long-term systems may handle such a problem better, but are inadequate to provide support for problem solving.

3.3.1.2.1 Advisor Scope

Providing expert assistance for a wide and complex process like software development, requires tackling scoping problems. A process may consist of several methodological steps, and cover different application domains, and the same user may need different kinds of support in different situations. An example of a scoping problem is, therefore, deciding when a change in the situation (step or domain) could enforce a change in the level of support provided to the user. As far as scoping problems are concerned, there is no systematic work on them in the literature. This is mainly because most of the attempts to provide expert assistance for software development focused on quite restricted domains, covering only one methodological step. The only "wide-scope" experiments, the Programmer's Apprentice and ASPIS, solve the problem by providing different advising systems for different life-cycle phases.

3.3.1.2.2 Passive Expert Assistance

Passive expert assistance refers to all support activities that a system tool can perform on request from a user; in other words, control of the dialogue between the user and the system is completely in the hands of the user. Passive expert assistance covers both common automatic support activities, such as automatic warnings and information retrieval; and complex knowledge-based activities, such as providing heuristics and advice on the task to accomplish, or giving an explanation of automatically performed operations.

It must be noted about advice-giving, that the dimensions of the problem change drastically depending on the level of support provided. Consider a passive assistant, driven by user queries, providing advice by consulting some knowledge-base, compared to a system which follows the user's actions and is able to suggest the next action to be performed at any time. This latter case requires knowledge-based activities, such as user monitoring and processing of knowledge about both development methods and problem domains.

3.3.1.2.3 Active Expert Assistance

When an expert assistant has the initiative, that is, it can interrupt the user to provide advice or training, it is considered active. Two main approaches may be adopted when providing active expert assistance:

a. Attention is focused on the goal of helping differently skilled users to perform the same task, providing guidance to them when necessary. The users' skills are raised as a biproduct.

b. The main purpose is providing enough general skill to a user so that the user can perform specific tasks afterwards.

Also, a mixed approach might be considered, providing tutoring to novices and guidance to users with a reasonable degree of expertise.

3.3.1.2.3.1 User Guidance

The term user guidance covers all expert assistance strategies where dialogue control is shared between the system and the user. The assistant takes the role of a supervisor who monitors the user's actions and, when diagnosing an error or even if there is some doubt or hesitancy, interrupts the user to provide support.

This support may take several forms. For instance, the system may simply provide information after an error is made or enter a confirmation dialogue before executing a "dangerous" action by posing questions that the user must answer before continuing (a trivial example: "Are you sure you want to delete all files in this directory?"). The former approach has had good results, especially with casual users who can recognize deadend situations better if they are allowed to meet them. When dealing with expert users, the opposite approach might be adopted. The assistant might automatically resolve incorrect inputs and suggest corrections to the user or, simply, make the corrections without requesting confirmation if the correction is obvious.

There is a hard requirement, common to all of these approaches: that is, in any moment of the session, the assistant must have a correct interpretation of user goals and intentions. To reach this, the assistant needs:

a. Methodical knowledge on the development task.
b. Domain knowledge on the application area.
c. A model of the user.
d. A monitoring strategy.

In addition to these functionalities, there should be:

a. Advisory approaches (see above).
b. Effective knowledge delivery to the user.
c. Natural language interface (possibly).

3.3.1.2.3.2 Computer-Based Tutoring

Tutoring systems are designed to provide instruction (on specific or general subjects) to unskilled users (ref. 3-4). Three different approaches to instruction may be identified:

a. Education: to provide conceptual principles that allow a person to think in abstract terms.
b. Training: to provide some theoretical information in the context of carrying out a particular procedure or accomplishing a specific goal.
c. Performance aid: to provide a minimal amount of training and emphasize providing a job aid.

As a consequence tutoring systems can be implemented following two different styles:

a. Socratic approach. Many tutoring systems employ a Socratic style where the system poses questions and the user provides answers. This approach offers little or no empirical rationale and leaves dialogue control to the system. For instance, after the user poses an initial question, the adviser assumes control of the interaction. Socratic advisers rarely include explanations or check whether the user understood the advice. Main criticism of such systems is that they "do not make the information meaningful".

b. Learning-by-doing approach. In this approach, the user can more freely initiate actions. Each user move is compared with an expert move generated by the system and feedback shapes the user's responses toward the expert prototype. This approach has the advantage of placing more initiative and, therefore, more control in the hands of the user. But, this flexibility requires more knowledge for the system which must know all the ways a user's action may depart from those of the normative expert.

To properly drive training, both these approaches present the problem of correctly evaluating what the user knows, does not know, or knows in a wrong way. People are not good at describing what they know, and especially what they do not know; therefore, the system has to infer this information from user's answers and actions. In synthesis, the main components of a tutoring system are:

a. Problem solving expertise: the knowledge that the system tries to impart to the user.
b. The user model: what the user does and does not know.
c. Tutoring strategies: how the system presents material to the users.

3.3.1.2.4 User Model

As elaborated in the previous section, expert assistance must guide a user through a G&C software development environment. For this purpose, the user modelling facility employs an expert model and a user or student model, where the expert model tries to mimic strategies followed by experts and the latter tries, if possible without a dialogue, to:

a. Recognize what he is doing.
b. Evaluate his goals.
c. Construct a model of what he is doing.

If the modelling facility is unable to determine these goals, it should try to acquire the necessary knowledge by starting a dialogue with the user.

The knowledge bases representing the expert and the current proficiency and expertise of a user, together with misconceptions and missing conceptions with respect to an expert are constituent parts of the user modelling facility. These bases are established by building mental states which describe the user's perception of the system, including his knowledge about the tools, the tool integration, the functionality of tools, the supported models, and the available knowledge bases, as well as all online help and tutoring facilities.

The user modelling facility closely relates to the active expert assistance facility as the computer-based tutoring component decides when and where to interrupt a user's work by analyzing his work and comparing the student with the expert model.

To provide individually and situationally pertinent advice, the system may need to analyze user activity. The development and maintenance of predictive user models is critical for advisory systems. Following are some possible approaches to developing predictive user models.

a. Normative models. A simple approach is to develop and refer to a single, normative model. For instance, the system might assume that new users of a package would not need to access specific advanced system functions; when users attempt to access these functions, they are informed that the functions are not available to them. Such an approach avoids user distraction on too advanced functions, but could result in user frustration should the user's actual goals fail to accord with the normative model. A mixed approach might employ a normative model with prescribed goals, but then adjust the advice dynamically in response to specific user actions.

b. Vocabulary analysis. A user's skill level might be determined by a key word analysis of the vocabulary employed in help calls. Each command should be associated with a hierarchical structure of explanations ranging from general to technical. The skill level diagnosed in the user's help call determines the vocabulary level of the explanation produced by the system. One problem with this approach is that it is not clear how diagnostic the vocabulary in a help call really is: an expert user might use relatively general vocabulary and, therefore, get advice at too elementary level. No behavioral assessment of the vocabulary analysis approach has been made yet.

c. Behavior analysis. Perhaps a more direct route to automatically diagnosing user skill is to monitor and evaluate the user's actual behavior. Much of this work has focused on monitoring and evaluating user errors. An approach to store typical error patterns as a normative user model: when particular users generate patterns of behavior that match a stored prototype, relevant corrective advice is produced. A more abstract level of analysis is based on knowledge "bugs", or systematic departures of a user model from an ideal model. A bug is defined as the smallest change in a correct procedure that would make it into an error. Unfortunately, research on behavior analysis is still quite inconclusive, since there is still no systematic theory of bugs or of error prototypes

d. Individual differences. A top-down approach to the problem could be to begin with a general analysis of individual differences and then deduce patterns of errors or bugs that are diagnostic. This advice-giving style represents the way humans provide information, and probably will be important in future advice-giving technology, but is undeveloped at present. The key problem is to identify the grain of analysis in order to distinguish between categories.

3.3.2 Involved Knowledge Bases

3.3.2.1 Expert Assistance

Active expert assistance is based on a variety of knowledge bases covering areas with a great impact on the development of guidance and control (G&C) systems. Examples are:

a. General methods and heuristics used in G&C development, covering concurrency, real-time behavior, continuous operations, resilience, and recovery. This knowledge base would contain "good" programming techniques and, maybe, the bad ones, to make a point.

b. Application and use of design or implementation methods, such as structured analysis or data modelling, using the entity-relationship-attribute approach. This knowledge base would contain "good" design techniques.

c. Support and explanation of the current technical procedure model valid for development of the whole project. In this case, the procedure model is the knowledge base.

d. Going beyond these three examples and taking expert assistance as a possibility to conserve technical and managerial skills, such as: project management, including personnel and cost management; or aerodynamics including, for instance, wing/airframe resilience tactics or electromagnetic compatibility.

Apart from these application-specific knowledge bases, temporal session specific knowledge about the current state of the system and the user with his expertise, personal profile and dialogue history must be available. Then, a model of the user's activity can be constructed, which is a prerequisite to evaluating the user's goals and plans.

3.3.2.2 Reuse

The process of identifying and retrieving reusable components is paramount. This could be done by matching the user's plans, his mental model, with appropriate components from the reuse library. This process is basically straightforward pattern matching with a certain 'metric' to identify components if there is not an exact match. Hence, finding a component suitable for reuse in a given application is not just locating an exact match. It may involve selecting the best choice from a family of components or finding a component that is similar to the desired one if it cannot be found. Even if similar components cannot be adapted, it may be worth finding them to be used as examples to develop new ones.

Components can have a very large set of reusability characteristics, which makes it difficult to assess their importance in the catalogue and retrieval processes. For this reason, the most promising approaches are based on AI techniques.

Pieto and Freeman (ref. 3-5) propose a classification method based on providing a number of attributes, or facets, for every component. A metric is used to measure conceptual distances between values in each facet. In this way, concepts from fuzzy set theory can be used to classify the components in a library. Wood and Sommerville (ref. 3-6) use ideas derived from natural language processing to understand the semantics of component descriptions or retrieval requests. The core of that approach is a number of fundamental concepts which are sufficient to capture the semantics in the domain of interest. Gerlach and Kuo (ref. 3-7) use semantic networks to catalogue modules according to their functions. However, there is no information about practical experience in using these approaches.

One of the key problems in reusing components is being able to find the components in a library that are most appropriate for a given application. Of course, the methods for retrieving components are closely related to the methods used to describe or catalogue components. A first approach makes use of database techniques, based on the entity-relationship or the relational models. The underlying concepts can be used to store and retrieve families of components where searches can be performed based on the desired attributes. Pieto and Freeman have suggested the use of fuzzy logic to perform component searching in a library. This is also related to the use of AI techniques, such as rule-based methods, for component retrieval. The latter approach has been demonstrated in the CAMP project (ref. 3-8).

Also, the ESPRIT project, PRACTITIONER, has focused on using AI techniques for classification and retrieval of reusable software components.

Along with intelligent identification and subsequent retrieval of reusable components, component construction also uses knowledge bases. A development environment needs to provide instruction and guidance to users about component construction through configuration, generation, adaptation, and available tools. This support means much more than simply furnishing the user with predefined templates. Certain programming methods, such as encapsulation, information hiding, modularity, or reentrance, must be enforced and deviations from these methods must be rejected.

3.3.2.3 Formal Methods

A commonly known program transformation system is a compiler. Recently, fifth generation languages, such as Prolog, have been used as design description languages with the added benefit that this design can be interpreted and executed. Further up the software development life cycle, reference 3-9 describes an expert system, which transforms a requirements specification (formalized in structured analysis, for instance) into a detailed design specification in a data flow representation.

The basis of these transformation systems is the formalization of design experience in a knowledge base, which can be manipulated further through production rules. Through a "recognize-act cycle" the existing structures of a requirements spec, are processed in steps and produce: data flow diagrams representing the actions to be performed by the systems; and a structured decomposition describing how these actions are implemented.

The latter actually is partitioning the overall system into modules. As an example, the REFINE method uses an object-oriented knowledge base for a system model and high-level language, using logic and set theory, for the specification of assertions on the knowledge base. Software is developed from the high-level executable specification (which provides the system prototype) into the target language, through a number of correctness-preserving transformations. The REFINE philosophy centers on the synthesis of executable code, currently LISP, from the high-level language which is executable and supports development prototyping (ref. 3-10).

3.4 APPLICABILITY IN SOFTWARE GENERATION AND AUTOMATED PROGRAMMING

Hardly any embedded computing systems have been developed from scratch. At least general systems engineering know-how has been used, if not parts of previous requirements, designs, or, even, implementation. Making software generation an engineering discipline implies that new systems will be developed by changing the already existing ones, thus, ensuring true technological advances.

With the advent of formal representation languages, automation of the software generation process becomes possible from the requirements specification to the executable code. Considerable progress has already been made in areas of straightforward data-base and business applications. Although we propose a possible automated process, it must be stressed that automated generation of highly complex guidance and control systems is and will stay a very creative activity.

A programmer's productivity, measured in lines of code per day, is almost independent of the programming language used. Therefore, radical new approaches must be taken such as:

a. Use of a very high-level language (VHLL) freeing both designer and programmer from the description of data structures.

b. Programming by determining commonalities between already implemented systems and new projects and iterative alterations which will fulfill the required specifications.

The use of VHLLs, which are by nature very powerful and extremely difficult, is limiting. The second approach is promising, since programming techniques, such as programming by example or fill in the form, have already emerged, at least for simple database applications.

The discussions in this chapter will concentrate on a formal description of a guidance and control system together with the executable code. All aspects of verification and validation as well as maintenance and generation of test sequences or documentation have to be discussed at a later stage.

3.4.1 Principles

This chapter proposes a general framework for automated programming, deduces some basic requirements, and tries to cover the most relevant steps toward a developmental environment. A top-level design, however, as well as a rationalization of the major stumbling blocks is left to a later stage. Although much of the following discussion may seem like fantasy, the main ideas are already beginning to materialize for simple business applications.

3.4.1.1 General Aspects of ECS Development

The prime objective of developing a computer-controlled guidance and control system is "building" a system that behaves per given requirements as shown in figure 3-3:

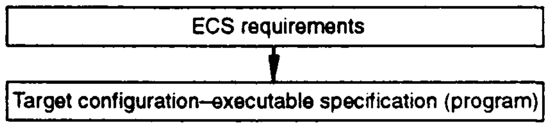


Figure 3-3. Principal G&C Development

Usually a computer-controlled system contains a computing system which may have several coupled computing systems and is connected to special devices. These devices are called "environment" to the embedded computing systems.

Also, a computing target may have several coupled individual targets and computer programs executable by the target processors. Hence, these computer programs can be seen as target-oriented "executable specifications" of the G&C system.

The immediate "transformation" of the G&C-related requirements into a suitable representation (that is, into a suitable target configuration including device coupling and associated executable specification) is restricted to very small and simple systems. The development of complex G&C systems needs a phased conceptual procedure.

3.4.1.2 Principal Conceptual Procedure of ECS Development

A phased conceptual procedure of the G&C development results from the fact that, generally, target-independent as well as target- and solution-dependent requirements and environmental characteristics can be distinguished. Any conceivable target configuration further implies target-specific constraints that must be regarded according to the following schema (fig. 3-4):

Item	Target-independent requirements	Target-and solution-dependent requirements
(1) Environmental characteristics	Logical • Signal and external • Data flow attributes	Physical • Signal and external • Data flow attributes
(2) System-specific requirements	Functional requirements • Functionality • G&C operation • Exception handling	Nonfunctional requirements • Performance • Safety • Security • Quality
(3) Target-specific constraints		G&C coupling target configuration • Device coupling • Operating system • Standard software • Development environment

Figure 3-4. Conceptual Approach

Consideration of the target-independent requirements allows the establishment of a target-independent G&C system representation, the so-called "essential model". The essential model includes the description of: regular behavior corresponding to the required functionality and G&C system operation; and fault-tolerant behavior to overcome exceptional situations.

The complete semiformal or formal description of the essential model, which may be derived from suitable abstract model parts (frames), is called a "specification".

The essential model must be adapted and extended so that the target- and solution-dependent nonfunctional requirements and the target-specific constraints are adequately obeyed. The resulting G&C system specification is then called the physical model.

The validation of the physical model and its transformation into a verifiable, modifiable structure (software-oriented design) and contents (implementation) of a target-oriented executable specification are the final phases of the G&C development conceptual procedure (fig. 3-5).

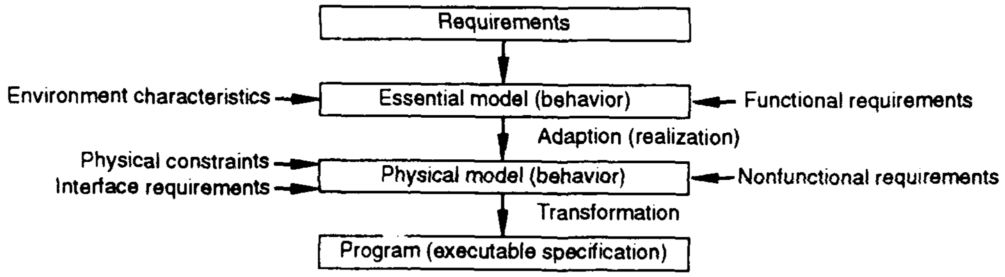

Requirements
Environment characteristics
Essential model (behavior)
Functional requirements
Adaption (realization)
Physical constraints
Interface requirements
Physical model (behavior)
Nonfunctional requirements
Transformation
Program (executable specification)

Figure 3-5. Principal Conceptual Procedure

Suitable formal languages for specification of complete, concise, and verifiable essential and physical ECS models are not available. Therefore, in most ECS development projects that have been completed, the essential model was established implicitly and incompletely in the context of the requirements analysis or by the help of semiformal specification techniques, such as structured analysis. The physical model usually was derived manually and specified by a suitable programming language backed by an ECS-oriented operating system. Therefore, the transformation to the executable specification was reduced to the generation of object code from source code.

3.4.1.3 Interactive ECS Development Based on Conceptual Procedure

The essential model that represents the logical behavior of a new ECS (the nominal essential model) must not start from scratch if a physical model of an existing related ECS (the actual physical model) is available. The actual essential model can be extracted from the actual physical model, and the nominal essential model can be derived from the actual essential model. This could be done through logical adaptation taking into account the difference of the nominal functional requirements and environmental characteristics from those of the actual ECS. Both the actual physical model and the actual essential model are abstracted using an actual system together with corporate know-how and experience.

Usually no formal specification of the physical model of the actual system is available except the executable specification. Therefore, it is necessary to abstract the actual physical model from the actual executable specification, that is, from the actual program. The successful utilization of the abstraction, extraction, and logical adaption procedure to create the nominal essential model and the nominal physical model depends on the availability of suitable formal specification languages. In this context, two different classes of specification languages (which are presently being investigated) may be distinguished.

a. Model-oriented languages, aiming at the specification of:
   1. Process- and function-oriented decomposition (hierarchy).
   2. State structures (hierarchy).
   3. Data object structures (hierarchies).
   4. Activation flow (net).
   5. Data flow (net).
   6. Event processing objects (abstract machines).
   7. Data transformation objects (abstract data types).

   Examples are:
   1. Structured analysis expanded by state transition diagrams.
   2. Predicate/transition net applications (Petri-nets).

b. Property-oriented languages, aiming at the specification of:
   1. Logical dependencies.
   2. Temporal dependencies.
   3. Facts and rules.
   4. Invariants.
   5. Preconditions and postconditions.

   Examples are:
   1. Extensions to predicate calculus (Prolog).
   2. Temporal logic.
   3. Interval logic.
   4. Modal action logic.

It seems that model-oriented languages are better suited for a graphic representation of a model that provides a better interface between engineer and computer within an ECS development environment.

However, property-oriented languages seem to be the proper means to establish an internal essential and physical ECS representation for automated conceptual procedures.

3.4.1.4 ECS Development Based on the Instantiation of a Generic Model

A first step could be made towards automated programming if adequate AI techniques could be exploited to automate, or at least support, the process of abstraction, extraction, logical and physical adaption, and transformation as indicated in figure 3-6. However, a modified approach, better suited to AI-technique application, is conceivable. The basic conceptual procedure can be generalized so that an arbitrary number n of actual systems (i=1,...,n), distinguished by related characteristics and obeying related requirements, is abstracted into n actual physical models and extracted n actual essential models. By a suitable combination of these models and by generalizing the environmental characteristics, system specific requirements, and target specific constraints into generic parameters, a generic model can be created. This model represents an ECS class characterized by the generic parameters.

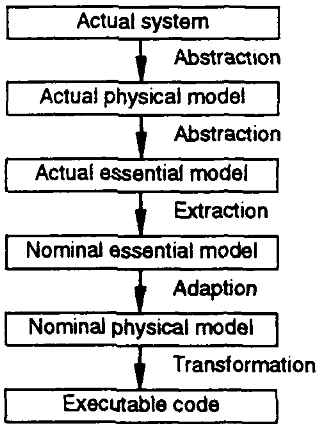


Figure 3-6. Abstraction, Extraction, and Adaption

Interpretation of the new specialized ECS environmental characteristics and functional requirements as actual generic parameters would then allow instantiation from a generic model, a nominal physical template of this ECS. Additional instantiation, together with the necessary physical adaption based on the target- and solution-dependent requirements and target-specific constraints, would lead to the nominal physical model of the required system.

If the nominal physical template or the nominal physical model cannot be fully instantiated because all nominal requirements cannot be allocated to generic parameters, the actual and essential physical models could be extended and an upgraded generic model created. The iterative modification of the nominal physical model and its template is done by adding new requirements and maybe even throwing away parts of the actual system. This process can be repeated until all nominal requirements are satisfied as shown in figure 3-7.

In the following section, the ECS development outlined procedure based on the instantiation of a generic model is discussed in the context of suitable AI techniques.

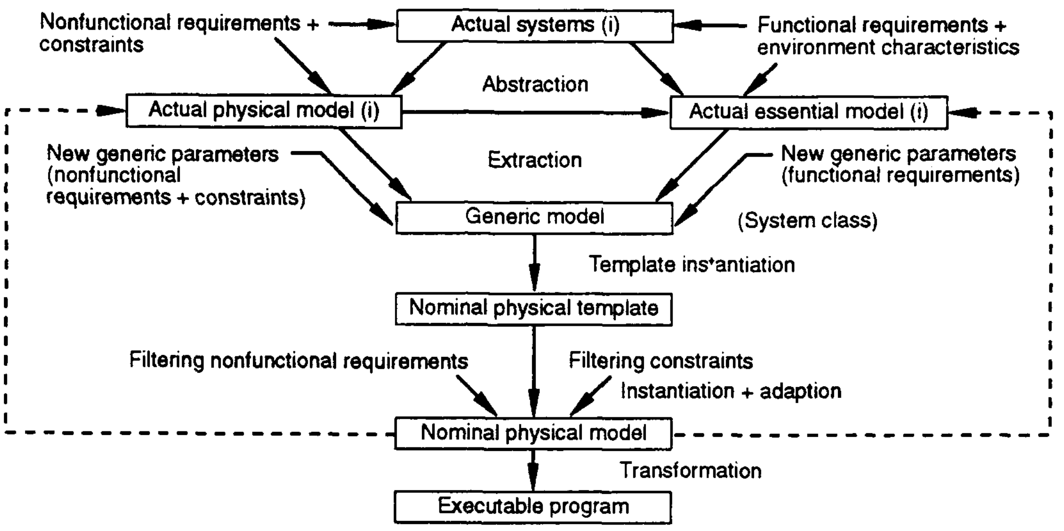

Nonfunctional requirements + constraints
Actual systems (i)
Functional requirements + environment characteristics
Abstraction
Actual physical model (i)
Actual essential model (i)
New generic parameters (nonfunctional requirements + constraints)
Extraction
New generic parameters (functional requirements)
Generic model
(System class)
Template instantiation
Nominal physical template
Filtering nonfunctional requirements
Filtering constraints
Instantiation + adaption
Nominal physical model
Transformation
Executable program

Figure 3-7. Iterative G&C Development

3.4.2 Abstraction of an Actual System Specification

Abstraction of an actual physical model from an existing implementation of a computing system is an iterative process reversing the development stages of that computing system. Hence, the problem is as follows:

a. Given: a guidance and control system implementation of an executable program written, for instance, in Assembler or Ada, together with an associated problem description and requirement specification.

b. Wanted: a physical model comprising a formal specification of computing system behavior and complete, though concise and noncontradictory, requirement specification.

Possible solutions to this problem are discussed in references 3-11 and 3-12.

3.4.2.1 Abstraction of Physical Models

3.4.2.1.1 The Programming Process

Provided the given problem has been strictly formalized (that is, both the specification and its design are written in a formal language) a programmer can start from the specification and produce, in an iterative way, the executable program. This iteration process is very machine-oriented as, in each loop, the specification is transformed into a more host-specific implementation language.

Optimization for code efficiency of the host implementation can even reduce simplicity and readability. On the other hand, new qualities are being added through this transformation.

These new qualities are: data and data structures; event structures and event processing objects (machine states); control and synchronization flows; and program structures.

Also, performance requirements may be added affecting the implementation issue even further. These constraints refer to: target and device coupling; the operating system, standard packages as drivers or supervisor calls; or even the development environment (restricted use of Ada for instance).

The result of this implementation process will be the embeddable or promable computer code which is also an executable specification or proof of the original requirement specification.

In general, the transformation process does not follow a linear graph, but, rather, a tree or an acyclicly-directed graph. Additional structure is being added with alternative implementations of the same specification. Using the contemporary methods of software engineering, all the knowledge of the bifurcation points (that is, different ways of implementation) in the design process may only remain for a short period of time in the heads of the related designers or programmers. This loss of information may seriously affect the product management and reduce the possibility of reusing parts of that particular product in subsequent projects.

3.4.2.1.2 Reversing the Programming Process

Starting from the implementation which is a machine-oriented specification of the embedded system, the formal specification has to be reconstructed. This reconstruction can be seen as a step-wise abstraction starting from machine-oriented source code and yielding a formal system description.

This can only be achieved using a knowledge-based system which employs knowledge about:

a. The programming language and appropriate standards, thus producing a first abstraction of the implementation.
b. The design methods yielding a design specification.
c. The original system requirements.

The second abstraction step gives the bifurcation points (i.e. certain design decisions) of the development process. This step will be to formalize the knowledge about design methods and knowledge about the original requirements that will be used by the system. The results of these abstraction processes are interactive with designer and programmer both using the implicit expert's knowledge.

This sequential abstraction involving both the control and data structures is shown in figure 3-8. (The abstraction of algorithms is left to a later time.)

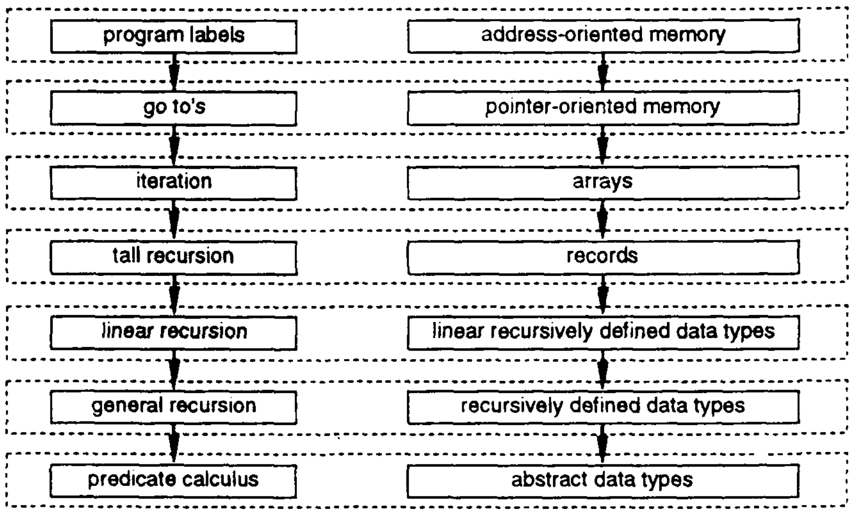


Figure 3-8. Control and Data Flow Abstraction

3.4.2.2 Extraction of Essential Models

Starting from an actual system and its deduced physical model, the application domain and the requirements of the new nominal system must be similar. It must be possible to use parts from a variety of generic models and, from this broadened base, develop a new system.

The creation of a formal description starting from an implementation has already been presented. Hence, a formalized actual system is available for further manipulation. It can be scrutinized interactively to add functional and nonfunctional requirements, constraints, and environmental characteristics. An expert system can actively support the developer in deducing the actual physical model and the associated essential model by using, as an inference mechanism, a forward-chaining recognize-act-cycle and, as knowledge bases, the formalized actual system; the corresponding requirements specifications; and general knowledge about the application domain, in particular environmental conditions or characteristics of the target platform.

3.4.3 Creation of a Generic Model Representing an Associated System Class

This section describes the creation of a generic model of the actual system and its derivatives, the physical and essential models.

3.4.3.1 Definition of the System Class

The system class is closely related to specific applications. The system class of a flight control system would embody all flight control systems. For a particular application, it is useful to limit the scope of possible applications and, for instance, restrict the class to jet fighter flight control systems.

Hence, the definition of a system class must contain:

a. The environment the application is supposed to operate in.

b. A precise description of the application domain.

c. A behavioral model.

Item c is part of the actual essential model and it already available. Items a and b, however, can be deduced from the new system requirements specification and general knowledge about the application domain, such as; operating and design principles and characteristics of embedded systems; and technical details of guidance and control systems together with flight characteristics, weight, wing/airframe resilience, etc.

The knowledge base must cover, in detail, the application domain so that an inference engine can operate through simple matching. The most likely knowledge representation scheme for this task seems to be the rule-based approach.

3.4.3.2 Transformation Into the Generic Models

A generalization of the environmental characteristics, system specific requirements, and target specific constraints of the physical and essential models yields the generic model of the actual system. Relinquishing the parameterization of the specific environmental characteristics and the implementation-specific constraints gives generic parameters. This generic model is the actual essential model without the specific references to nonfunctional requirements of the actual system. Together with the description of the system class, however, the qualitatively new generic model is created.

Besides substituting specific parameters with generic parameters, the structure of the generic model may change as well. From the formal representation of the essential model general rules, commonalities, connections, and structural similarities can be induced to create new objects and their corresponding hierarchy. In principle, this restructuring process could be done automatically using inductive expert systems. But the problem that may arise is that the automatically-created generic model still represents the same system class.

3.4.3.3 Visualization of the Generic Model

With the introduction of generic parameters, the generic model representation becomes difficult to understand, let alone manipulate further. Hence, the generic model must be transformed into a different representation using graphic constructs and icons which visualize the generic model and alleviate the manipulation process. General induced rules and structural similarities, together with their relationships, are described by integrating text and icons. This must be done interactively leaving the whole denomination process of creating a template of the generic model to a user.

It must be ensured that this "iconization" is a one-to-one mapping of the generic model because, in later steps, the graphic representation must be recompiled into the formal representation to allow an iterative development process.

Once a first template exists, it can be manipulated and animated introducing functional requirements and environmental characteristics of the nominal model. Thus, the generic model representing the original actual system can be transformed into a generic model of the nominal system.

3.4.4 Iterative Derivation of a Nominal System

The nominal system includes the nominal essential system and the nominal physical system. It is derived from the generic system by an instantiation process taking into consideration all requirements.

3.4.4.1 Instantiation of the Nominal System Specification

In the notations of deMarco (ref. 3-13), the nominal system represents the nominal essential model of the system. It contains on a high level of specification, all information that is essential for the system planned.

3.4.4.1.1 Essential Models in Software Engineering

Per reference 3-13, there are four basic principles for the creation of essential models :

a. Degree of complexity: the limited human ability to grasp the complexity of a problem must be taken into account. If the essential model is too complex, a person cannot understand the specification and, therefore, cannot check its accuracy.

b. Technological neutrality: the essential model should not contain any information about the system implementation.

c. Perfect internal technology: no assumptions about the properties and behavior of the system environment should be made.

d. Minimal essential models: there are several ways of modelling an essential requirement, but the most simple one should always be chosen.

These principles apply to and give hints for the derivation of the nominal essential model from the generic model.

3.4.4.1.2 Essential Models in Guidance and Control Applications

Basically, there is little difference between embedded systems in G&C applications and other systems. But the stringent safety requirements together with hard real-time requirements for G&C applications impose some additional requirements on the model representation which must be satisfied. These two attributes of G&C applications must be covered explicitly by the expressive power of the model representation to allow for semiautomatic analysis of these attributes by an expert system.

If one looks onto the wide area of G&C related applications it is not likely that the whole field is covered by a single representation language, for example:

a. Flight guidance, missile guidance: properties of these kinds of applications are simple algorithms and complex structure.

b. Radar, navigation: properties of these kinds of applications are complex algorithms and simple structure.

The safety argument enforces a further distinction of G&C systems to other applications. This distinction is that all modifications to algorithms enforced by physical limitations in the implementation (e.g., word length effects in numerical computations) must be treated in the nominal essential model because they may influence the system safety. Consequently, some features which, in a classical approach, belong to the physical model must be included in the essential model.

3.4.4.1.3 Physical Models

A physical model gives a behavioral (functional) description of the required system and addresses specific implementation points such as exception handling, performance or safety and security levels, as well as hardware/software definitions. The following list defines the necessary functions to be performed with the physical model:

a. Adaption of models. In the following steps, the main task will always be to adapt a model to some requirements or to transform a model on a certain level of abstraction and formalization to a model with another level of abstraction and formalization or with another representation. This adaption process and its possible support by AI methods is described in the following paragraphs. Two of the main AI paradigms, the model-based and learning approach, are the kernel of the adaption procedure. Hence, the procedure may be supported by an expert system and it is inherently AI.

b. Mapping internal onto external view. To enter requirements into a model and to modify the model adequately, it is necessary to have a connection between the internal model (the model to be implemented) and the external view of the model (the requirements imposed by the user).

Two views of the model must be distinguished: the external view describes what is done by the system; the internal view describes how it is done.

The notation of views must not be mixed up with the notation of a model representation. The representation used by a computer system (internal representation) will be different from the representation of the model to the user of the system (external representation). Moreover, in an expert system, there are several layers of representation going from a high-level representation (e.g., symbols, nets, grammar, with associated semantics) down to the byte-and-bit level.

c. Views in a common model. For the explanation of the following transformation steps, we use a rather abstract and mathematical description which gives deeper insight since it intuitively parallels classical cybernetic ideas. We look at models and specifications as elements of an abstract space in the same way as an isolated feedback control law may be represented by the elements of its system and control matrices.

In this model, combining the internal model S and the external views by modelling a property, requirement, or constraint is mapping the system S to its properties $e_1,...,e_n$ using the statements fn(S) = en where en lies in some nontrivial space or set E. In a system-theoretic view, E can be seen as the output-space of the system S consisting of the target system and its environment. (This must not be mixed up with the I/O behavior of the nominal system).

A model interpreter may derive fn(S) from the model S without the need to implement S on a real machine (e.g., by semantic interpretation or numeric simulation).

d. Requirements comparison. The first step in the transformation from requirements to model modifications is the comparison and assessment of the differences between the actual properties of the model and the required properties stated in the requirements.

e. Deriving modifications. The conclusion drawn from the difference in the external model (properties) to the difference in the internal model is a highly heuristic one based on experience and system knowledge.

An approximate solution relating differences to differences is in a primarily mathematical model based on the following regularity and linearity assumptions:

1. Regularity: differences in the external model (result, output in the system theoretic view) can be traced down to differences in the internal model.
2. Linearity: the conclusion from differences in the external model to differences in the internal model is at least approximately or with differences known to the expert independent from the absolute values, and depends only on the differences.

From the regularity and linearity conditions, some kind of continuity follows, (i.e., small differences in requirements or properties can be removed by small modifications in the system).

f. Analogy. To find a similar model or an analogous difference is not an easy task for a computer-based system. It needs some assessment, evaluation, or metrics on the set of all models.

g. Sequencing. In general, a lot of selections and modifications have to be made to match the system properties with the requirements. Since these modifications influence each other (nonlinearity), an additional iterative cycle is necessary. But the system can only converge, if the sequence of modifications is intelligently chosen.

Starting from the independent modifications followed from model or result differences, refined modifications have to be derived taking into consideration the dependencies between all modifications. After having performed all modifications, some iterative loops of assessment and modifications have to be made in which either the original modifications are slightly changed or resequenced or new modifications have to be added.

3.4.4.2 Generation of a Nominal System Template

The structure and representation of the nominal system template (the nominal essential model) is inherited from the generic model. Hence, the generation of a nominal essential model is mainly an instantiation task. This necessitates a very flexible generic model with a powerful representation mechanism.

The main issue in the instantiation process is not the instantiation of a generic system to special parameters, but the selection of those parameters per the special requirements.

3.4.4.2.1 Instantiation and Parameter Selection

Instantiation is the process of selecting one element or subclass from a class of elements. For the software generation process, this means that we select a system implementation (an actual program or a template) from a class of possible programs. This class is given by a generic system and the instantiation of the generic system is done by parameter selection. There is a continuous range of possible parameters to be used for generic model instantiation ranging from:

a. Flags and switches.
b. Single real numbers like filter constants and timing constants.
c. Order relations like priority levels or timing relations.
d. Single numbers like word length.
e. Complex data structures like lists, matrices, or trees.
f. Data structures as an object.
g. Relations and instructions expressed in a 5th generation programming language.
h. Sets of structures and data.

3.4.4.2.2 Knowledge-Based Parameter Selection

Parameter selection from the requirements is a challenging tasks which needs a lot of knowledge and heuristics. In today's application examples, this is restricted to the selection of a set of (in general real) numbers. But the steps indicated above may lead parameter selection to a form of software generation, hence, converging parameter selecting expert systems and automated programming.

3.4.4.3 Instantiation of a Nominal Physical Model

3.4.4.3.1 The Nominal Physical Model

The nominal physical model is built on the basis of the nominal essential model by incorporating real-world aspects and constraints. It is now relevant to analyze the effects of those constraints on the performance and the behavior of the nominal essential model in order to guarantee the satisfaction of system requirements.

3.4.4.3.2 Nominal Physical Model Representation

Models cannot be built only on one main aspect of the system, but must consider the structure of the system as well as its behavior. A possible model class for representing these models is outlined in the following:

a. Hierarchical multiaspect networks: networks are a natural way of representing structure and interrelations. Since many multihierarchy aspects must be modelled, network models have to consider communications between processing objects and the relations between them. This may be done by using several networks with different semantics such as:

   1. Communications: data and control, structured analysis, state-transition-diagrams.

   2. Relations and facts: entity-relationship-diagrams, semantic nets.

   3. Physical model dynamics: dynamics cannot be derived only from the above networks, so further representations are necessary including network dynamics (nets, petri-nets, state-transition-diagrams) and temporal logic (temporal relations between events, timeliness, scheduling).

3.4.4.3.3 The Transformation Approach

Looking at the derivation of physical models as a transformation, this step is covered by the model modification. The model modification approach will be covered in section 3.4.5.

3.4.4.3.4 The Design Process

The step from the nominal essential model (template) to the nominal physical model parallels the step from the generic model to the template. The requirements entering the model at this stage are non-functional and constraints which differ mainly in the pragmatical content but are of similar semantics and of the same syntax as the functional requirements entered at the previous step (under the assumption of a strong representation language for requirements). From the transformation systems point of view, this does not make much difference, hence, the same remarks and constraints as in the previous chapter apply.

The main differences are:

a. Decisions in the Instantiation Process. Rather than making one large decision step, the design (instantiation) is done by many steps of design and modification decisions.

b. Modification versus innovation. Systems engineering and development as a whole (and to a much greater extent, those parts covered by expert systems and automation) are evolutionary rather than revolutionary. Automated systems can make improvement modifications rather than innovative development. This allows the reuse of models, specifications, and code.

3.4.5 Iterative Adaption to the Nominal Requirements and Constraints

At any step of the process of generating and testing the nominal model (i.e., in the instantiation of the nominal physical model and the nominal essential model, as well as in the implementation phase) it may become evident that the generic model is inadequate or too restrictive. In all these cases, an iteration cycle has to start that modifies the generic model in order to comprise all features necessary in the nominal models.

Generic model modification is the most intellectually challenging task in the whole software development cycle since structural modifications and deep modelling of the problem become necessary at this point. Expert systems may support this process if there is a greater pool of possible generic models from which a new or adapted generic model can be taken. This comes from the fact that the generic model (as well as any essential model) is designed for some minimal complexity in order to make formal treatment possible.

Expert systems may also point out a weakness in the generic model and show (e.g., analogies to differences in other models), hence, giving a hint to the human model builder how to modify the generic model or even the generic model pool.

3.4.5.1 Classes of Generic Models

The class of generic models is the set of all mathematical models which is too large a set to be covered by any computer representation. From this it seems to be clear that a selection and a restriction has to be made for those models in the generic systems pool. Moreover, the generic systems must be classified by several criteria in order to have a way of retrieving the needed model. Possible classes of generic systems might be:

a. By the target: guidance, navigation, and decision support.
b. By the data: deterministic models, noisy models, and game theoretic models (counterpart, deception).
c. By the representation: symbolic models and numerical models.

Other classification criteria involve the representation of time axis (we have assumed that G&C models are dynamic), target model, and representation and description level.

3.4.5.2 Intelligent Model Modification

Selection and combination of generic models from these classes may be supported by expert systems which know about the specification of the generic models and are able to match them with the nominal requirements. Anything that has been said on model modification is also valid for the modification of generic models.

Tracing problems back to models by analogy involves an iterative cycle of the following steps:

a. See the analogy between the problem P and a problem P' that is already solved. Analogy implies some metric on the set of all problems.
b. Find the difference between P and P'.
c. Find a pair P1 and P1' of already solved problems, for which the difference is similar or analogous to the difference between P and P'.

d. Find the difference in the problem solution (i.e. in our case in the model or the specification) between the solution S1 to P1 and S1' to P1'.

e. Transform the difference and apply this modification to the solution S' to P' and hope that the modified solution S will solve P.

The remarks about regularity and linearity made previously are valid to a larger and more critical extent for this kind of meta-problem-solving.

3.5 LIMITATIONS

3.5.1 Validation and Verification

Validation is the process of evaluating software at the end of the software development process to ensure compliance with software requirements. Verification is the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. This process encompasses the formal proof of program correctness including the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items conform to the specified requirements.

In this section, we will concentrate on the validation of software, although the principles discussed apply as well to the other life cycle phases. Validation in the early engineering phases, such as the preliminary and definition phases, can be done on a basis of checking adherence to a given form and completeness of referencing the requirements. The validation process will give an idea about the correctness of software. This should not be mistaken with debugging, as this tries to localize errors starting from a known deficiency of the product. Validation consists of a test and a test strategy.

A test describes a finite subset of the set of all possible values a program can work with. From the definition of this finite set the correctness for the whole domain is deduced. A test strategy defines the way to construct these subsets.

In practice, several test strategies are existing which, in some parts, differ extremely in efficiency and effort required. So, the problem is to choose the right strategy or combination of strategies to meet the reliability and security requirements of the software and to reduce the effort to a tolerable amount. More details about verification, validation, and testing together with possible problems are in references 3-14 and 3-15.

3.5.1.1 Verification and Validation of Tools

Validation and verification (V & V) of the system which supports the software generation is very important. The verification of the generated software can be done in different ways (i.e., verifying all generated software parts explicitly or verifying the generating system and, in doing so, verifying the generated software by transformation). It is much more difficult to verify the generating system than to verify the developed software because the software for guidance and control applications is less complex. But it may be better to verify a complex system once to save verifying all the developed systems. Verification of the expert system includes:

a. Proof that the system transforms specifications into an executable form. It must be guaranteed that the result of the transformation is executable.

b. Proof of completeness. All possible structures of a specification must be detected and transformed. This syntactical completeness may be easy to prove; much more difficult is to guarantee semantic completeness. The rule must transform the whole semantic content of a structure.

c. Proof of correctness. It must be guaranteed that the right transformation rules are selected (not only referring to the syntax but also to the semantic content of a structure) and all the conditions for application of the transformation rules are proved. If several rules are possible, they must have a priority order and selection criteria must exist.

d. Proof of applicability of the system. The system must be applicable in practice and work for more than very simple problems.

3.5.1.2 Validation and Verification of Expert Systems and Knowledge Bases

Within the framework of V & V, a heuristic system cannot be formally proven. Otherwise, we have a formal system which in some sense contradicts expert systems paradigms. Building an expert system is different for conventional programming. The expert system is developed in interaction with a domain expert. The knowledge engineer must acquire knowledge from the expert, aid the domain expert in structuring knowledge, and formulate rules. One approach to this verification is V & V of the inference engine and the knowledge base.

Instead of validating and verifying the results of the requirement phase, design phase, implementation, and integration which is usual in conventional software development, the validation and verification during the building of an expert system should be divided into the following steps:

a. V & V of the procedure.

b. V & V of an expert system shell (if applicable).

c. V & V of the knowledge base.

The procedure in building an expert system is different than conventional programming. The expert system is developed in interaction with a domain expert. The knowledge engineer must acquire knowledge from the expert, aid the domain expert in structuring knowledge, and formulate rules.

The expert system shell must be appropriate to the problem. For supporting the reusable-software approach, another tool may be appropriate than the tool for supporting the program-transformation approach because the knowledge may be represented in a different way. The domain concepts and control structures must be represented with the chosen tool.

The knowledge base may become very large. So validation and verification of the knowledge base is important to guarantee that no fundamental concept is omitted. In a rule-based environment for instance, priority, consistency, and redundancy have to be covered. It is inherently much more difficult to do V & V on a frame-based or hybrid shell. Methods still have to be evaluated.

3.5.2 Knowledge Acquisition and Updating Strategies for Knowledge Bases

Knowledge acquisition and structuring of the knowledge into a usable form are the most important and time-consuming tasks in developing knowledge-based systems. The knowledge may be extracted from several sources, such as reports, case studies, personal experience, etc. However, in most cases the human expert is the main source of information and eliciting an expert's knowledge is not a one-step linear process, but takes interaction between a knowledge engineer and domain experts over a long period of time. One difficulty in capturing knowledge is that there are no algorithmic approaches. Instead, the expert's knowledge is heuristic, subjective, and not necessarily organized. Because of the subjectivity, it is advisable not to fill the knowledge base with knowledge acquired from only one domain expert. However, this knowledge may be contradictory in some cases because different experts have different procedures in problem solving. (Some techniques of knowledge acquisition are described in reference 3-16.)

Updating strategies for knowledge bases are much more complicated than those for databases, because integrity constraints are inherently difficult to formalize and, thus, enforce. This is a consequence of the dependencies between rules or objects of a knowledge base where an assertion or retraction may have an impact on the behavior of the overall system. Knowledge bases supporting any engineering process must evolve in the sense that they can be updated or whole new knowledge bases incorporated. Hence, they must be easy to modify. (Experts which have to be told everything in minute detail are not experts.) Currently, there are no automatic ways on the horizon supporting the knowledge-acquisition process, though machine-learning techniques may mature in the near future. In the meantime, a more conservative approach could be to use a structured method to gather feedback between the user and knowledge engineer responsible for the intrinsic knowledge bases and the relevant information about new cases. The knowledge bases can then be upgraded manually.

Because of the spatial distribution of knowledge bases, other problems have to be addressed, such as access of different people to knowledge that may be unclassified, Restricted, Confidential, Secret or, even, Top Secret. For classified data contained or processed in a knowledge-based system, the following security information has to be processed:

a. The security markings of every piece of information. This may include information on the system's own configuration.
b. The security level and privileges of every authorized user of the system.
c. The security level and privileges allowable in every location where there is an instantiation of the system.
d. The security level and privileges allowable on every piece of equipment (ports), connected to the system which may communicate with the outside world (i.e., outside of the local development environment).

Apart from these security considerations aimed at protecting company- and project-specific know-how, there are also additional consideration to handling authorization and personal access to data, such as:

a. Data security from corruption, failure and recovery.
b. Concurrency control.
c. Ordinary integrity control.

There considerations have to be addressed for open and distributed development environments. The obvious approach is to use expert databases or deductive database mating aspects from the database side and the knowledge-base side.

3.5.3 Expert Databases

In theory, there are two ways of integrating databases with expert system technology. The loose coupling is done through an interface module which manages facts, rules, or objects in the database. Integrating database predicates with the expert systems manipulation language is called tight coupling. The advantage of tight coupling is that the access to the database is transparent to the user. Both models are applicable for relational and object-oriented data models. The advantages of these deductive databases are that they enable: constraint propagation, distributed AI systems, and knowledge-base management.

Constraints describe restrictions of admissible states of a data model. Rules within the database manipulation language allow the checking and modelling of functional dependencies of objects. Rules also allow the implementation of triggers and active values which are the basis for a dynamic data model. These features are quite handy for version control and configuration management. Distributed AI systems are necessary if the environment itself is distributed and various systems access time-independent facts, rules or objects. In those cases, the expert database is used not only to ensure integrity and concurrency control, but is also a communication system using blackboards. These blackboards are used to integrate and extract information (knowledge) from different sites.

In database technology, the database administrator performs a well-known role. These functions are also applicable in knowledge-base management ensuring well-structured knowledge representation, optimization of data storage, and definition of the correct semantic theory which is the baseline for the knowledge representation technique.

3.6 FUTURE R&D

Several ways for supporting development of embedded computing systems by expert systems have been discussed. Some points remain open as they are still being researched. Some open points are:

a. Formal specification: the formal specification process covers only the transformation of functional requirements into a specification. This document describes possible ways to incorporate nonfunctional requirements. However, to make this approach operational much more effort is needed.

b. Advanced user interfaces and representation methods: as discussed, automated programming requires sophisticated graphic representation techniques that are easy to animate and manipulate. Methods and techniques do not exist to do this.

c. Prototyping: currently, the process of rapid prototyping is not incorporated into G&C development model. To explore efficiently the advantages of prototyping in all development phases, a prototyping method, as well as appropriate tools for rapid prototyping must be developed.

d. Verification and validation: only the problems of verification and validation have been discussed. However, a great deal of work is needed to develop techniques and methods.

3.7 REFERENCES

3-1 A Survey and Classification of some Program Transformation Approaches and Techniques. Information Sciences Institute, 1986.

3-2 G.E. Kaiser, P.H. Feicer, S.S. Popovich: "Intelligent assistance for software development and Maintenance", IEEE Software, Vol. 5, No. 3, pp. 40-49, May 1988.

3-3 R.C. Houghton: "Online Help Systems: A Cospectus", Communication of the ACM, vol. 27, pp. 126-133, February 1984.

3-4 E. Wegner: "Artificial Intelligence and Tutoring Systems", Morgan Kauffmann Publishers, Los Altos 1987.

3-5 R. Pieto-Diaz, P. Freeman: "Classifying software for reusability", IEEE Software, January 1987.

3-6 M. Wood and I. Sommerville: "An information retrieval system for software components", SIGIR Forum, Vol. 22, No. 3-4, pp. 11-28, 1988.

3-7 J. Gerlach and F. Kuo: "Auser interface design for cataloging and identifying reusable software modules: a semantic network approach", Proc. of the 22th Annual Hawaii International Conference on System Sciences, Vol. 22, pp. 1035-1040.

3-8 C. Anderson: "The CAMP approach to software reuse"; Defence Computing, Sept.-Oct. 1988.

3-9 Intelligent Support for Specification Transformation IEEE Software, November 1988.

3-10 REFINE: Refine User Guide, Reasoning Systems Inc. 1987.

3-11 M.T. Harandi, J.Q. Ning: A Knowledge-based Program-Analysis; IEEE Software, Vol. 7, No. 1, pp. 74-81, January 1990.

3-12 W. Kozaczynski, J.Q. Ning: "SRE: A knowledge-based environment for large software reengineering activities", Proceedings of the International Conference on Software Engineering, CS Press, Los Alamitos, California, pp. 113-122, 1989.

3-13 Mc Menamin and Palmer: "Strukturierte Systemanalyse", Hanser Verlag 1988.

3-14 G.J. Myers: The Art of Software Test; Oldenbourg-Verlag, 1987.

3-15 W.J. Quirk: "Verification and Validation of Real Time Software", Springer-Verlag 1985.

3-16 D.A. Waterman: A Guide to Expert Systems; Addison-Wesley, 1986.

Additional references for the current status of research:

R. Bell, H. Atchan, R. St. Dennis: "A toolkit for development of Database-centered Applications", Proceedings fourth International Conference on Data Engineering, Golden Valley, USA 1988, IEEE Comput. Soc. Press, ISBN 0-8186-0827-7.

R. Balzer: "A 15 year perspective on automatic Programming", IEEE Transactions on Software Engineering, Vol SE11, pp. 1257-1268, Nov. 1985.

D. R. Barstow: "An Experiment in knowledge-based automatic programming", Artificial Intelligence, Vol 12, No. 2, pp. 73-119, 1979.

F. L. Bauer et al.: "The Munich Project CIP, the Wide-Spectrum Language CIP-L", Vol.1, Springer Verlag, LNCSS-183, 1985.

F. L. Bauer et al.: "The Munich Project CIP, The Transformation System CIP-S", Vol. 2, Springer Verlag, LNCS-292, 1987.

B. I. Blum: "Experience with an automated Generator of Information Systems", International Symposium on new directions in Computing", USA 1985, IEEE Comput. Soc. Press, ISBN 0-8186-0639-8.

B. Cousin, P. Estrailer, M. Kordon: "Generation of Ada code from a PETRI net model: an application", In Proceedings of the 3rd International Symposium on Computer and Information Sciences, Oct. 1988.

Cleveland: "Building Application Generators", IEEE Software, July 1988.

R. N. Cronk et al.: "ES/AG: An Environment for the Generation of Expert Problem-Solving Application Systems", Conference on Software Tools, New York, USA 1985, IEEE Comput. Soc. Press, ISBN 0-8186-0628-2.

M. Demail: "Automatic generation of industrial Databases from PROLOG", CIIAM 86, Proceedings of the 2nd International Conference on Artificial Intelligence, pp. 555-566, France, 1986.

C. Dixon: "The Use of the Application Generator COMUNIX '85", Proceedings of EMAP Conference, pp. 6-13, London 1985.

F. S. Fickas: "Automating the transformational development of software", IEEE Transactions on Software Engineering, SE-11, No. 11, pp. 1268-1277, 1985.

B. Gates: "Gentran: An Automatic Code Generation Facility for Reduce", Sigsam Bull. (USA), Vol. 19, No. 3, pp. 24-42, 1985.

N. K. Gupta, R. E. Seviora: "An Expert-System Approach to Real-Time System Debugging", Proceedings First Conf. Artificial-Intelligence Applications, CS Press, Los Alamitos, California, pp. 336-343, 1984.

L. L. Gremillion, T. Shea: "Cobol Application Code Generators", J. Syst. Manage. (USA), Vol. 36, No. 12, pp. 30-33, 1985.

E. Horowitz, A. Kemper, B. Narasimhan: "A survey of application generators", IEEE Software (USA), Vol 2, No. 1, pp. 40-54, 1985.

M. T. Harandi, J. Q. Ning: "A Knowledge-based Program-Analysis Tool", Proc. Conf. Software Maintenance, CS Press, Los Alamitos, California, pp. 312-318, 1988.

S. Honiden, N. Uchihira, H. Nakamura: "Automatic Programming for Control Systems", Int. Process. Soc. JPN, Vol. 28, No. 10, pp. 1398-1404, 1987.

E. Kant: "A knowledge-based approach to using efficiency estimation in program synthesis", In Proceedings of the 6th IJCAI, Stanford, California, 1979.

R. Keller, P. Towsend: "Knowledge-Based Systems", Computerworld (USA), Vol. 18, No. 49A, pp. 31-32, 1984.

P. A. Luker, A. Burns: "Program Generators and Generation Software", Comput. J. (GB), Vol. 29, No. 4, pp. 315-321, 1986.

F. Nishida, Y. Fujita: "Semi-automatic Program Refinement from Specification using Library Modules", Transf. Inf. Process. Soc. JPN, Vol. 25, No. 5, pp. 785-793, 1984.

J. Q. Ning, M. T. Harandi: "An Experiment in Automating Code Analysis", Proceedings AAAI Symp. Artificial Intelligence and Software Engineering", AAAI Press, Stanford, California, pp. 51-53, 1989.

J. Q. Ning: "A Knowledge-based Approach to Automatic Program Analysis", doctoral dissertation, University of Illinois at Urbana-Champaign, 1989.

S. Nakata, G. Yamazaki, G. Ohishi: "Database-Oriented Application software Generation System", NEC Techn. J., Vol. 40. No. 1, pp. 39-45, 1987.

H. Parisch, R. Steinbruggen: "Program transformation systems", ACM Computing Surveys, 15(3), 1983.

H. Sneed, G. Jandrasics: "Inverse Transformation of Software from Code to Specification", Proc. Conf. Software Maintenance, CS Press, Los Alamitos, California, pp. 102-109, 1988.

D. Wile, R. Balzer: "Automated derivation of program control structure from natural language program descriptions", SIGPLAN Not. 12(8), 1977.

D. Wile, R. Balzer, N. Goldman: "On the transformational approach to programming", In the proceedings of the 2nd ICSE, San Francisco, California, 1976.

CHAPTER 4

PROGRAM TRANSFORMATION APPROACH TO SOFTWARE GENERATION

4.1 OVERVIEW

In the beginning of the 70s, the software development task was structured in the same way people develop planes. The most famous innovation was the waterfall model proposed by B. Boehm (ref. 4-1). This model introduced the notion of "steps" during which a specific work is conducted. The set of steps and their precise order has been clearly identified. Each step is followed by verification and validation (V&V) which ends either by the normal continuation of the process (the next step), or by going back to the previous step to correct errors that were discovered.

Unfortunately, all the errors, bugs, mistakes, and failures are not always detected by the V&V processes. Moreover, the later the errors are detected, the more expensive is their repair.

Motivated by the increasingly significant cost of software development and maintenance in the computing environment, researchers have sought methodologies for program development (ref. 4-2). One such methodology that has been advocated is that of program transformation. The application of program transformation is not limited to assisting software development. Other areas include:

a. Investigating classes of algorithms (i.e., synthesis of several algorithms from a single specification, etc.)
b. Assisting program description and verification (i.e., establishing proved programs).
c. Adapting existing programs (e.g., adaption and maintenance of programs).

As part of AGARD/GCP/WG10, focusing on software generators, we will mainly concentrate on the software development application of program transformation in this chapter.

The program transformation approach is a methodology.

4.1.1 Program Transformation Role

Per reference 4-3, for any program transformation methodology, where it emphasizes "synthetic" process of deducing the program from its formal specification or a more "analytic" process of deducing properties from the already finished program, pure creativity goes hand in hand with many automatable activities. Program development already has substantial support if the automatable parts are performed by a machine and the programmer is free to concentrate on the creative aspects.

The program transformation approach should be a mechanizable methodology.

4.1.2 Program Transformation Definition

The word "program" is to be read in its broad sense: a *program* is a description of computation that is expressible in a formal language. A *program scheme* is a class of related programs: it originates from program by parameterization. Programs, conversely, can be obtained from a program scheme by instantiating the scheme parameters. In its most general form a *transformation* is a relationship between two program schemes P and P'. It is said to be *correct* (or valid), if a certain semantic relationship holds between P and P'. Thus, a transformation rule is a mapping from program to program.

According to the kind of relationship (transformation), we will get several kinds of properties for the transformations, such as those preserving the semantics of the initial program, and those giving a weak semantic to the transformed program.

Transformational programming is a methodology of program construction by successive applications of transformation rules. Usually this process starts with a formal[1] *specification* (that is, a formal statement of a problem or its solution) and ends with an executable program. The individual transitions are made between the various versions of a program by applying correctness-preserving transformation rules. It is guaranteed that the final version will still satisfy the initial specification.

The program transformation approach is a constructive and mechanizable methodology.

4.1.3 Program Transformation Domain

We can look at transformations as follows:

a. At the same level of language, program transformation appears as program modification:

1. If *inputs* are *programs* (in the classical sense) then transformations act as a general support for *program modification.* Program modifications include either *optimization* of control structures, the *efficient implementation* of data structures, or both.
2. If *inputs* are *program schemes* then transformations act as *generators* of *new transformation rules* (correct by construction).

b. At different levels of languages, program transformation is more concerned with transformation belonging to the following domains:

1. *Program synthesis* is a transformation starting from a specification (or from examples or mathematical assertions) and ending with the generation of a program.
2. *Program description* is also a transformation which allows exhibit of the derivation built up for a program.
3. *Deduction-oriented verification* of program correctness is also a part of the domain of transformation. Let us remember how mathematical proofs are made!

4.1.4 Transformation Types

The transformation rules may be of the following types:

a. *Procedural rule* which has the form of an algorithm, taking a program and producing a new one.

b. *Schematic rule* (or pattern replacement rule) which is expressed as a statement of the form:

$f(x_1, \ldots, x_n) \Leftrightarrow g(u_1, \ldots, u_p)$ expressing the equivalence of an input (leftmember) and an output (rightmember) program scheme. Using such a rule, each possible instantiation of the left program scheme in the program to transform is replaced by the corresponding instantiation of the right program scheme. The equivalence between these two schemes may be strong (both schemes have the same semantic properties) or weak (going from left to right, specific properties are added, thus weakening the generality of the semantics.)

c. *Hybrid rules* such as FOLD and UNFOLD.

[1] *Formal means: syntax and semantics well defined by a precise mathematical notation.*

Usually the procedural rules are called *global*, schematic ones are called *local*. The global rules (also called semantic rules) have more to do with global treatments such as consistency checks, cleanup operations, representations of programming paradigms (for instance, divide-and-conquer). The local rules correspond to basic transformations such as:

a. L: if B then S; goto L fi ⇔ while B do S od

   This transformation will be used to optimize all the sentences (of a PASCAL-like language) matching the left-hand side by replacing them by the right-hand side. The optimization is both from the point of view of readability (better program structure) and from the point of view of efficiency (GOTO will disappear).

b. $b \wedge b \Leftrightarrow b$ which in logic-like language is called the idempotence property (if b is of boolean type).

Specific attention must be paid on the FOLD and UNFOLD rules. They are very often used in transformation systems and imply peculiar techniques for their use. Indeed, UNFOLD is the replacement of a call (or interface) by its body (or expansion) with appropriate substitutions. In other words, a pattern-matcher is needed. Moreover, because several substitutions hold, a decision must be taken on the unique substitution to be done. This decision depends a lot on the kind of system we wish to use. Fully automatic, fully interactive (decisions are taken step by step by the user) or intermediate (with, for instance, the presence of a metalanguage for directing the decisions). FOLD is the inverse transformation (expansion or body of code is substituted by an interface or a call).

4.2 GENERAL TRANSFORMATION SYSTEMS

This state of the art description is not established in chronological order. First, it presents the general transformation systems; then, some specialized ones. Another state of the art description about projects is presented later.

4.2.1 General Approaches

In this section, we discuss the research in which transformations were studied in depth, either per se or as a basis for the development of environments.

4.2.1.1 The ISI Approach

At the Information Science Institute of the University of Southern California at Marina Del Rey, several systems have been developed. Among them we can cite Specification Acquisition From Experts (SAFE) (ref. 4-4) and Transformation Implementation (TI) (ref. 4-5). SAFE deals with the synthesis of formal specifications from informal ones. TI concentrates on the derivation of efficient programs from formal specifications. Many other transformational systems are under development. POPART (ref. 4-6) also considered applying transformations to program developments (ref. 4-7).

4.2.1.2 The PSI System

The PSI system (ref. 4-8) is a large system including several subsystems, such as PMB, PECOS (ref. 4-9), LIBRA (ref. r-10). PSI was designed to synthesize efficient programs from specifications obtained earlier. The PECOS system is a coding expert which contains a list of 400 transformation rules. The LIBRA system is the efficiency expert that gives advice to the coding system. The studies conducted around PSI have been continued with the CHI system which is now marketed as the REFINE system.

4.2.1.3 The Edinburgh School

Burstall and Darlington conducted their work to the definition of hybrid rules (ref. 4-11). Among the rules DEFINITION, FOLDING, UNFOLDING, INSTANTIATION, ABSTRACTION and LAWS constitute a solid (i.e., theoretical and pragmatic) basis for a lot of transformation systems. A number of important examples have been treated according to the simple but powerful systems derived from their approach.

In the reference example, it is shown how, from a formal (mathematical) specification of a problem, it is possible to derive several algorithms just by using transformations. The example is about *sort*. The formal specification is restricted to the mathematical expression of *sort* (e.g., a set of combinations of permutations). Depending on the criteria used, the resulting algorithms range from *bubble sort* to *quick sort*, *heap sort*, etc.

The reference example is also important because it clearly shows that, at the top level of the formal specification phase, only functional properties of the software to be produced are expressed. Nothing is said about the constraints (or nonfunctional properties). Darlington's approach shows how the nonfunctional properties are taken into account during the transformation of specifications. ZAP (ref. 4-3) is also involved in the approach of Burstall and Darlington.

4.2.1.4 The Stanford Approach

Manna and Waldinger have developed the DEDuctive ALgorithm Ur-Synthesizer (DEDALUS) system (ref. 4-12). Using transformations, this system's aim is automatic production of LISP programs. The objective is more proof oriented.

4.2.1.5 The Arsac's Approach

Arsac's system was designed to interactively manipulate imperative programs (ref. 4-13). One of the main results is the automatic transformation of recursive procedures into imperative ones (ref. 4-14).

4.2.1.6 The CIP System

The Computer-Aided, Intuition-Guided Programming (CIP) project was conducted at Technical University of Munich by Bauer and his team (refs. 4-15, 4-16, and 4-17). CIP, from the point of view of transformation, manipulates program schemes of which concrete programs are a special case. The main interest of CIP relies on the manipulations that can be done on the schemes. Indeed they are concise (one scheme can be instantiated into several concrete ones) and allow the use of contexts (parameterization of rules).

4.2.2 Specialized Transformation Systems

In this section, examine specialized systems to which the transformational approach has been applied as a well-improved technique.

4.2.2.1 MENTOR

MENTOR is a syntax directed editor. Its main purpose is to help users in the documentation, implementation and transportation phases (ref. 4-18). This system transforms any external syntax into a single abstract representation called Abstract Syntax Tree (AST). A set of tools can act on the AST of any program by using transformations. Some of them concern the propagation of modifications within a program, some others are related to the transformation into another external syntax.

4.2.2.2 SPRAC

SPRAC Systeme de Programmation par Reutilisation Assistee de Connaissances (SPRAC) is a software engineering environment that helps users from specification to code production (SPRAC) (refs. 4-19 and 4-20). SPRAC allows the transformation of formal specifications expressed in LF (an assertional strongly typed language based on the first order predicate calculus) into LA (an abstract algorithmic language) and then to LM (the classical programming language).

The transformation between LF and LA is semiautomatic. Indeed, as with the Darlington approach, part of the transformations are automatic (for instance, the transformation of recursion into imperative structures), but part of them must be directed by the user.

The transformation between LA and the programming language is fully automatic (ref. 4-21) and has been shown as being optimal. In this context, we can affirm that the *transformational approach* is a *methodology*.

4.2.2.3 The VDM Approach

The VDM approach (refs. 4-22 and 4-23) is in this same line of development method but:

a. It includes the specifications (preconditions, postconditions).

b. It does not provide a possibility of executing the first high level of program development. (It does not, thus, provide any specific support to prototyping.)

4.2.2.4 Other Systems

The list of comparable systems is very long and should be updated in the near future.

4.3 TRANSFORMATION OF SET THEORETIC DATA STRUCTURES INTO LOWER LEVEL DATA STRUCTURES, EFFICIENTLY HANDLED BY THE CLASSICAL IMPERATIVE LANGUAGES: AN EXAMPLE

4.3.1 Introduction

This section is directly extracted from a paper written by P. FACON (ref. 4-24) and deals with program transformations which are used in two important projects: RAPTS (ref. 4-25) and ESPRIT/SED, both aiming at translating set theoretic languages into the usual imperative programming languages such as Ada.

By their expressive power, set theoretic primitives are a preferred tool for formal specification and, when they are executable, for fast prototyping. However, sets have not machine representation which is efficient in the general case. Only a fine analysis of the set operations of such a specification may allow one to select efficient representations.

This section presents program transformations performing such a task. The set theoretic language which is considered here is SETL (ref. 4-26) and its derivates which are used by both the above mentioned projects. However the same category of program transformations is used as well in other systems such as REFINE (ref. 4-27).

4.3.2 The SETL Language and the RAPTS Transformation System

Designed at New York University in the mid-70s, SETL offers a complete set of primitives to handle finite sets. Beyond sets, the other handled objects are tuples and maps (maps are functions which graph (set of couples input/results) is finite and given by enumeration).

To illustrate SETL style, we can say that besides classical set operators as the set of parts "powerset", SETL also offers as specialized set operators as the set of parts having exactly "k" elements of a set "s": "k npow s" which may also be written "s npow k" since SETL has a very flexible syntax. SETL offers also the definition of sets by comprehension from a basic set ($x \in E|C$), existential and universal quantifiers and iterators, the possibility to extract a random element from a set, the notions of subsets, cardinality, etc.

Furthermore, SETL is a classical imperative and sequential language. Typing is extremely dynamic and no declaration is compulsory. SETL is, thus, very well tuned for fast prototyping. Another very important research axis around SETL has always been program transformations.

R. PAIGE and his team have designed a specification language SQ+, which is a small functional subset of SETL and can be seen as a "high-level" SETL augmented by fixed point definition, together with an associated transformational system, RAPTS (ref. 4-25). The aim of this system is to automatically generate efficient imperative programs by using a set of transformations starting from an SQ+ specification.

We will extend our presentation a bit more on SQ+ and, after that, we will take a look at the set machine language (SML) language which is used an an intermediate language between SQ+ and the classical imperative languages. SML can be seen as a "low-level" subset of SETL.

SQ+ is a set theoretic specification language which has the following main characteristics:

a. SQ+ is purely functional, without assignment and iteration over sets.

b. The main objects are finite sets, maps (altogether sets of couples and mono or multivalued functions), and tuples.

c. Beyond the classical set theoretic operators, SQ+ offers operators to give minimization (respectively maximization) or least (respectively greatest) fixed point definitions:

   1. Themin s: $s \geq w|c$ is the smallest set s larger than w for which c holds.
   2. lfpw(s, e) is the smallest set s larger than w such that s=e. It is generally preferable to use the fixed point notation rather than the minimization/maximization notation since lfpw(s, e) can be transformed, given some additional conditions into the following iteration:

      s :=w;

      (while $\exists$ a $\in$ e(s)-s) s:=s $\cup$ {a} end

SML has the following characteristics:

a. SML is an imperative language handling the same objects as in SQ+.

b. The only set theoretic operations are elementary operations.

c. An operation like ID with :=X where an element X is added to the set ID has always $X \notin ID$ as a precondition (i.e., the precondition must be explicitly tested before performing the operation is one cannot assert it holds).

The main expressions and primitives of SQ+ and SML are displayed in figures 4-1 and 4-2. In choosing an implementation model for sets, two subproblems arise:

a. Which sets will be effectively implemented as lower data structures?

b. How will this implementation be performed?

Concerning a, the first idea is to implement exactly those set type variables which are present in SQ+. However, this will generally not be enough. Some set type expressions in SQ+ would be costly to implement

$S \cup T, S \cap T, S - T$	union, intersection, difference
$S \subseteq T, S = T$	inclusion and equality tests
#S	cardinality of S
$\exists S$	gives a random element of S
$X \in S, X \notin S$	membership tests
$\{\}, \{X1, X2, \ldots, Xn\}$	empty set, enumerated set
$\{E(X) : X \in S \setminus K(X)\}$	set defined by comprehension
$\forall X \in S \setminus K(X), \exists X \in S \setminus K(X)$	universal and existential quantifiers
domain F, *range* F	map domain and image
F(X)	image of the element X through mapping by F (should this image be unique)
$F\{X\}$	set of images of X
$F[S]$	set of images of elements of S
lfpw(s,e)	least fixed point of λs.e greater than w
lfp(s,e)	shorthand for lpfØ(s, e)
gfpw(s,e)	greatest fixed point of λs.e smaller than w
themins : $s \geq w \setminus c$	least s greater than w making c hold
themaxs : $s \leq w \setminus c$	greatest s smaller than w making c hold

Figure 4-1. SET Theoretic Expressions in the Language SQ+

$\exists S$	random element of S
$X \in S, X \notin S$	membership tests
$\exists X \in S$	existential quantifier (with a side effect: gives X the value of a random element of s when S not empty)
F(X)	image through a map (undefined if not unique)
$F\{X\}$	set of images

(a) Expressions

ID := {}	initialization of an empty set
ID*with* := X	strict addition of an element
ID*less* := X	strict subtraction of an element

(b) Assignments

-sequence, alternative, iteration as in classical imperative languages
-iteration over a set: (for $X \in S$) INST end
Let us notice that SETL includes among others all the SQ+ and SML operators except the minimization/maximization and fixed point operator of SQ+)

(c) Control Structures

Figure 4-2. Main Primitives of the Language SML

as expressions and new variables must be introduced to store their result, which must also be implemented as lower data structures. We will give further details on this. New variables are introduced during the phase of translation from SQ+ to SML. The other result of this translation is the replacement of all the global set theoretic operations by elementary operations (i.e., operations which are performed item by item). As an example, the SQ+ operation:

A-B (difference between two sets) is translated into SML as:

C:=Ø; (for X ∈ A) if X ∉ B C with := X end.

The basic techniques to solve b will be developed after the techniques for introducing new variables.

Combining both a and b, the global implementation scheme is represented in figure 4-3.

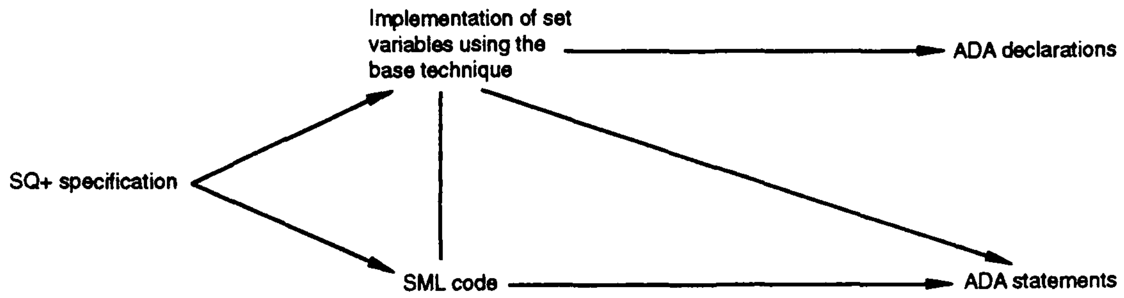


Figure 4-3. Global Implementation Scheme for SQ+ Specifications

4.3.3 Optimizing Set Computations by the Introduction of Intermediate Variables and Finite Differencing

Two of the main transformation techniques used in the RAPTS system are the search for fixed points and optimization by differencing.

In optimization by differencing, the basic idea is to let EXP be a set theoretic expression which is costly to compute and is repetitively evaluated within some iteration process. Then, introduce an auxiliary variable E and maintain the invariant E=EXP at each point where EXP is evaluated.

To maintain the invariant E=EXP, we will have to:

a. Compute EXP and assign its value to E the first time EXP is evaluated.

b. Actualize the value of E each time EXP is evaluated with some modification in an actual parameter by a "differential code" of EXP with the modified parameter.

Here are two examples of such "differential code":

a. Let us suppose we have to maintain the invariant:
C=card A where the parameter A is the object of a "strict" addition: A with:=Z (asserting Z was not previously in A)
the differential code for C=card A is then:
C:=C+1

b. In the same conditions as in a, if we have to maintain the invariant:
D:={X ∈ A |K(X)}
the differential code will be:
if K(Z) then D with:=Z.

This optimization is valuable only if the cumulated cost of the differential codes remains smaller than the cost of recomputing the expression. Cost computations are then performed by induction on the structure of expressions.

On another side, the technique is applied to any subexpression of a complex expression; invariants will have to be maintained and differential codes produced in cascades.

In the following example, implementation involves both techniques (search for fixed points and optimization by differencing with the introduction of an auxiliary variable):

a. Informal requirement for our example:

Let us consider a graph given by a set of edges: succ, taking the form of a map relating the nodes and their successors, and let w be a subset of the nodes.

Compute the set r of nodes reachable from the nodes belonging to w.

b. Formal SQ+ specification:

```
r=themin s | w⊆s and succ[s]⊆s
```

c. Reformulation of the SQ+ specification using a fixed point:

```
r=lfp(s,w∪succ[s])
```

d. Using a transformation (from section 4.3.2), this SQ+ fixed point specification can be transformed into:

```
r:=∅
(while ∃ a ∈ w ∪ succ[r]-r)
r:=r ∪ {a}
end
```

e. Introducing the invariant t=w∪succ{r}-r, this is further transformed into:

```
t:=w
r:=∅
(while ∃ a ∈ t)
t:=t ∪ succ {a}-{a}
r:=r ∪ {a}
end
```

f. The final SML code is then obtained from this point by unfolding the global set operations into iterations using only the elementary set theoretic operations of SML; this transformation yields:

```
t:=∅
(for x ∈ w) t with := x end
r:=∅
(while ∃ a ∈ t)
(for y ∈ succ{a})
if y ∉ r and y ∉ t then t with :=y end
end
t less := a
r with := a
end.
```

The transformation techniques we just described are implemented in the RAPTS system and are also partially implemented in the ESPRIT project SED, with the aim of developing a software environment around SETL. The SED environment incorporates also a translator from "low-level SETL" (similar to SML) to Ada.

4.3.4 Base Technique: Choosing Efficient Representations for Sets

Potential implementation for sets are numerous and different implementations lead to very different algorithms. This is both an advantage (freedom in the implementation of sets) and a drawback (it raises the problem of choosing the best implementation).

An often-used default implementation for sets is the double-link list with a hash table. The hope with this implementation is that it should support both traversing the set and performing an associative access to one of its elements. The aim of an efficient implementation of sets is to minimize altogether:

a. The cost of traversing sets.

b. The cost of associative access to individual elements of sets.

c. The data redundancy: a value which is shared by several sets should be stored only once.

The default implementation which is mentioned above fulfills item a well, item b poorly, and item c not at all. The basic idea of the optimized implementation we are proposing to fulfill all three aims is as follows:

An associative access to a set Q (resp. to the domain of a function F) is avoided if the value X of the element which is the target of the access is stored at an address from which the address of the value of the logical expression $X \in Q$ (resp. of the expression F(X)) can be directly computed without using the value of X itself. Then both the access time and the memory space are optimized since the value X is stored only once.

Let us consider as an example the iteration:

(for $X \in E$)...if $X \in S$...end

where we say we have a "production of X by E" and an "access to S through X".

The aim is to have a unitary cost for the access to S. Let us build an array of records B where each recorded element X has a field X.value and a boolean field X.S being TRUE if $X \in S$ and FALSE otherwise. Let E be implemented as a list of record elements in B. The iteration is then implemented by a simple iteration over the list E with an access to the field X.S for each element.

We have also minimized the memory space occupation since each value X.value is stored only once despite the fact that X may belong to two different sets E and S. We say that B is a "base", S is "strongly based on B", E is "weakly based on B".

If we complicate our example of expression by adding some traversing of S, then we will implement S as a list of recorded elements in B and make the field X.S be a pointer, which is null if $X \notin S$ and which refers to the next element in the list implementing S if $X \notin S$, rather than be a simple boolean equal to $(X \notin S)$. To illustrate this "base technique" for implementing set variables, we can reuse the graph example in section 3.3. Figure 4.4 shows a copy of the final SML program obtained in section 4.3.3 to which we added comments classifying the set theoretic expressions as "productions" or "accesses".

```
t := Ø
(for x ∈ w) t with := x end
r:= Ø
(while ∃ a ∈ t)                    - - production of a by t
   (for y ∈ succ {a})              - - access to domain (succ) through a
                                   - - and production of y
       if y ∉ r and y ∉ t then     - - access to r and t through y
           t with := y
       end
   end
   t less := a                     - - (self) access to t through a
   r with := a                     - - (self) access to r through a
end
```

Figure 4-4. The SML Program Implementing the SQ+ Specification

The analysis of this program, enriched by the classification of set theoretic expressions, leads to the following conclusions for a lower level implementation of this program into an Ada program (as an example):

a. A unique base B should be generated having one record element for each item (node of the graph) belonging to w ∪ domain[succ]∪range[succ] (which turns out to be equal to the set of nodes of the whole graph).

b. On this base B, r, t, domain[succ] should be strongly based (i.e., to each should correspond to a dedicated field in the elements of B which could be named "∈ t", "∈ r", "∈ domain[succ]") since associative accesses are performed on them, and range[succ] should be weakly based on B since it is only traversed.

The set t is the object of two kinds of operations:

a. Associative access (tless, twith).

b. Production of an element from t (while ∃ a∈ t).

The elements of t in the field "∈ t" should then be linked by pointers. The expression "(while ∃ a∈ t)" leaves freedom at the implementation level in the choice of element a, should it exist. The best choice is to take it to the end of the list of pointers implementing t because the access to the end of the list is convenient as well as for the other operations "twith", and also "tless" since tless is used as "tless:=a" being embedded in "(while ∃ a∈ t)...tless:=a end" and can, thus, be simplified as "renaming the last element of the list" (which was chosen in the implementation of "(while ∃ a∈ t)") instead of being "removing the element a".

So finally the "∈ t" field will be implemented as pointers pointing to the index of the preceding element in the list, and we will add to that a data "last element of t" giving an immediate access to the last element of the list without being obliged to search it through all the elements of B. The field "∈ r" is a simple boolean being TRUE if "i∈ r" holds (i being the index of an element of B) and FALSE otherwise since only associative accesses (rwith, y∉ r) are performed on r. The field "domain[succ]" is a pointer to a list containing references to all the corresponding (by succ) elements in "range[succ]" (since range[succ] is weakly based on B). Figure 4-5 gives an example of a graph together with the scheme of its based implementation as it was just discussed.

Considering the Ada graph interface of figure 4-6 the body of function IS_SUCCESSOR can be directly developed as an implementation of the SML program of figure 4-4 using the base we have just defined and which is schematized in figure 4-5.

Figure 4-7 displays this Ada implementation intermixing the SML code as comments inserted in the Ada code (an SML statement always precedes the Ada code fragment it generates). But, contrary to this very simple

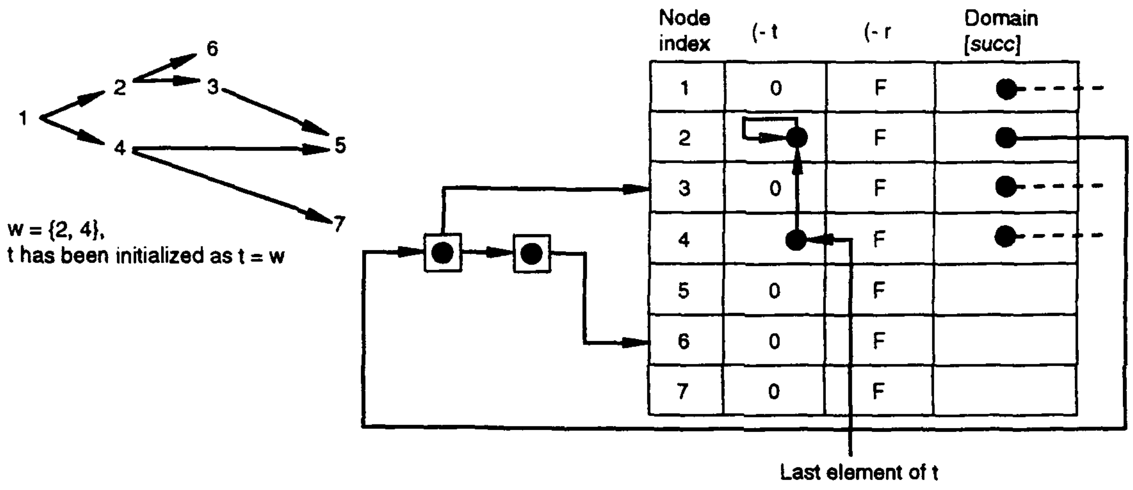


Figure 4-5. An Example of a Graph and a Scheme of its Based Implementation

```
package body GRAPHS is
  function IS_SUCCESSOR (G : GRAPH; NODES : NODE_SET) return NODE_SET is
    type T_FIELD_ARRAY is array (NODE_INDEX) of NODE_DESIGNATOR;
    type T_FIELD is
      record
        LAST_ELEMENT : NODE_DESIGNATOR := 0;
        SET : T_FIELD_ARRAY := (others =¿0)
      end record;
    type BASE is
      record
        G : GRAPH;
        R : NODE_SET := (others =¿FALSE);
        T : T_FIELD;
      end record;
    B : BASE;
    procedure T_WITH (I : NODE_INDEX) is
    - - (t with := x)SML where x is the node of index I in G;
    begin
      if B.T.LAST_ELEMENT = 0 then–t is empty
          B.T.SET (I) := I;
      else
          B.T.SET (I) := B.T.LAST_ELEMENT;
      end if;
      B.T.LAST_ELEMENT := I;
    end T_WITH;
    procedure T_LESS is
    - - removes the last element of t;
    - - ASSERT : B.T.LAST_ELEMENT /= 0;
      LAST_ELEMENT : constant NODE_INDEX := B.T.LAST_ELEMENT;
    begin
      B.T.LAST_ELEMENT := B.T.SET (LAST_ELEMENT);
      B.T.SET (LAST_ELEMENT) := 0;
    end T_LESS;
  begin
    B.G := G;
    - - t :=∈
    - - (for x∈ w)
    for I in NODE_INDEX loop
      if NODES (I) then–t with := x
        T_WITH (I);
    - - end
      end if;
    end loop;
```

Figure 4-6. An Ada Graph Interface

```
with SINGLE_LINK_LISTS;
generic
    NODE_NUMBER : POSITIVE;–number of nodes in each graph
package GRAPHS is
    subtype NODE_DESIGNATOR is NATURAL range 0 .. NODE_NUMBER;
    - - 0 designates no node;
    subtype NODE_INDEX is NODE_DESIGNATOR range 1 .. NODE_NUMBER;
    package NODE_INDEX_LISTS is new SINGLE_LINK_LISTS (NODE_INDEX);
    use NODE_INDEX_LISTS;
    subtype SUCC_LIST is LIST;
    type GRAPH is array (NODE_INDEX) of SUCC_LIST;
    - - each element of the array defines the direct successors of the node having
    - - the corresponding index;
    type NODE_SET is array (NODE_INDEX) of BOOLEAN;
    - - the TRUE elements designate a subset of the nodes of a graph;
    function IS_SUCCESSOR (G : GRAPH; NODES : NODE_SET) return NODE_SET;
    - - returns the set of nodes of G which are reachable form the nodes of G
    - - in NODES;
    - - (r = IS_SUCCESSOR (G,W))ADA = W
    - - (r = themin s |W  s and succ [s] ⊆ s)SQ+;
end GRAPHS;
```

```
generic
    type ITEM is private;
package SINGLE_LINK_LISTS is
    type LIST is private;
    EMPTY_LIST : exception;
    procedure APPEND_TO (L : in out LIST; I : ITEM);
    - - POSTCONDITION :
    - - A new element is added to the last position of L with the value I;
    function IS_EMPTY (L : LIST) return BOOLEAN;
    - - returns TRUE if L is empty and FALSE otherwise;
    procedure GET_LAST_ITEM (L : in out LIST; I : out ITEM);
    - - PRECONDITION :
    - - not IS_EMPTY (L);(otherwise EMPTY_LIST is raised)
    - - POSTCONDITION :
    - - (I receives the value of the last item of L) and
    - - (the last item of L is removed from L);
private
    basic_declarative_item...
end SINGLE_LINK_
LISTS;
```

Figure 4-6. An Ada Graph Interface (Continued)

example, several bases have elements that can be based sets. A base then corresponds to a union of sets communicating values one to another.

Since the "dynamic" bases (with values that can be changed at run time) have proved themselves to be rather slow to handle, we will consider "static" bases, which are constituted from elements which are initially present. The sets which cannot be based on such static bases will be given the default implementation: double-link lists, with hash tables if the set is associatively accessed.

By analyzing the set theoretic operations of an SQ+ or SML or SETL program, its bases can be computed together with the implementation of each set on the bases. This generates the Ada data structures and the Ada code is generated from these data structures and from the SML code as shown in the previous example. Knowing the translation of the global set theoretic operators of SQ+ into SML code, it is possible to analyze directly the SQ+ specification and the invariant definitions rather the SML code. For example, the SQ+ set difference operator: A-B has the following translation into SML:

C:=Ø; (for X ∈ A) if X ∉ B C with := X end

```
    - - r :=
    - - (while ∃ a∈ t)
    declare
      A : NODE_DESIGNATOR;
      Y : NODE_INDEX;
    begin
      while B.T.LAST_ELEMENT /= 0 loop
        A := B.T.LAST_ELEMENT;
        - - (for y∈ succ a)
        while not IS_EMPTY (B.G (A)) loop
          GET_LAST_ITEM (B.G (A),Y);
          - - if y∉ r and y∉ t then
          if not B.R (Y) and (B.T.SET (Y) = 0) then--t with := y
              T_WITH (Y);
            - - end
          end if;
          - - end
        end loop:
        - - t less := a
        T_LESS;
        - - r with := a
        B.R (A) := TRUE;
        - - end
      end loop;
    end;
    return B.R;
  end IS_SUCCESSOR;
end GRAPHS;
```

Figure 4-7. Package Body GRAPHS Generated From the SML Code

This implies that B is strongly based and A is weakly based on the same base. This rule can then be directly associated to the SQ+ set difference operator.

4.4 CURRENT STATUS

Much of the current work in program transformations is being conducted in the ESPRIT project. This work is a continuation of works started on transformations several years before.

4.4.1 PROSPECTRA

4.4.1.1 Project Contents

The aim of PROSPECTRA (program development by specification and transformation) was to gather a set of methods and tools for a formal development of programs obeying the following life cycle:

a. Requirement specifications:

   1. Nonconstructive (property-orienteu,
   2. May be loose, at least when first established.
   3. PROSPECTRA offers a language to write such specifications and verification tools, but it does not offer any tool to help establish them from informal requirements and system analysis.
   4. The PROSPECTRA project team considers that these specifications must be negotiated with the client who must be convinced that they fulfill his requirements. They become a "formal contract" for the subsequent part of the development.

b. Design specifications:

   1. Constitute an "abstract implementation" (model-oriented).

2. PROSPECTRA offers program transformations to go from requirement specifications to design specifications.

c. Efficient program construction through transformation:

   1. PROSPECTRA offers program transformations with some aids to handle them.
   2. Programs are finally generated in the Ada language, but other target languages are also foreseen.

The program development is entirely historized to allow a replay (presently, only a pure replay is possible, but a possibility to modify the replayed program development is introduced). The language used during all the development is called PANNDA. It is a derivative from Ada and AnnA with algebraic and functional extensions (the sublanguage used for specifications, which is the only one to be directly handled by the PROSPECTRA user, is called PANNDA-S).

The languages TRAFOLA (used to describe transformation schemes and their environment) and CONTROLA (used for the control of the development system) are themselves PANNDA sublanguages.

4.4.1.2 Project Origins

PROSPECTRA has inherited from different sources:

a. The CIP project of the University of Munich (refs. 4-16 and 4-17) has given the general transformation approach, a number of transformations, and a part of its development team.
b. The language AnnA, annotating Ada programs with specifications and assertions, and one of its main designers, B. Krieg-Bruckner who lead the PROSPECTRA project.
c. The Cornell synthesizer generator which has been used as the implementation kernel of PROSPECTRA tools.

4.4.1.3 Project Team

Many universities took part in the project: the Universities of Bremen (prime contractor), Dortmund, Passau and Saarland in Germany, and the Strathclyde University in England. A few industrial companies are also involved: Syseca Logiciel (F), CRI (Dk), ALCATEL Standard Electrica SA (E).

4.4.1.4 Specification and Transformation Examples

The examples in this section all deal with an abstract data type (ADT) LIST which provides the necessary material to build and handle lists of elements of any items. A first example of a requirement specification for such an ADT is given in the package LISTS in figure 4-8. From this first requirement specification example, we can follow the development cycle of the function "length". The abstract implementation phase, leading from the requirement specification to the design specification, will leave the function "length" unchanged ("length" is defined in terms of more basic operators and its implementation will not use the implementation of type LISTS which is the main object of the transformations performed during this phase).

In the next phase of development, the applicative implementation, the user wants to implement "length" using an algorithm which may still be recursive, but which can be transformed later into a nonrecursive more efficient one. To perform such a transformation, the transformation scheme in figure 4-9 will be used. The resulting implementation of "length" is given in figure 4-10.

```
generic
   type ITEM is private;
   "<": ITEM→ ITEM→ BOOLEAN:: for all x, y, z: ITEM
                              ⇒ not(x<x),
                                 x,y and y<z → x<z;
package LISTS is
   type LIST is private;
   empty: LIST;
   cons: ITEM → LIST → LIST;
---------------------------------------------------------------
   "&": LIST → LIST → LIST;
   single: ITEM → LIST;
axiom:: for all x, y, z: LIST
          ⇒ x & empty = x,
             empty & x = x,
             (x & y) & z = x & (y & z);
---------------------------------------------------------------
   isempty: LIST → BOOLEAN;
   head: (x: LIST:: not(isempty x)) → ITEM;
   tail: (x: LIST:: not(isempty x)) → LIST;
axiom:: for all e: ITEM, l: LIST
          ⇒ isempty empty = true,
             isempty (cons e l) = false,
             head (cons e l) = e,
             tail (cons e l) = l;
---------------------------------------------------------------
   "<=": LIST→ LIST→ BOOLEAN;
axiom:: for all l, l1, l2: LIST; x1, x2: ITEM
          ⇒ l <= l & l2,
             x1<=x2 → 1 & (single x1) & l1<=1 & (single x2) & l2;
---------------------------------------------------------------
   length: LIST → INTEGER;
axiom:: for all x: LIST
       ⇒ isempty x → length x = 0,
          not(isempty x) → length x = length(tail x) + 1;
---------------------------------------------------------------
end LISTS;
```

Figure 4-8. A First Example of a Requirement Specification

Input Scheme and App. Conditions	Output Scheme
f: S → M; axiom:: for all x: S => ¬ B x→ f x = f (H x)⊕ X x, B x → f x = T x;	f: S → M; g: S → M → M ; axiom:: for all x: S => f x = g x n, ¬ B x→ g x y = g (H x)((X x)⊕ y), B x→ g x y = (T x)⊕ y;
such that *t does ¬ occur in* T, H, X axiom:: for all x, y, z: M => (x ⊕ y) ⊕ z = x⊕ (y ⊕ z), x ⊕∩ = x;	

Figure 4-9. Transformation of a Linear Recursion Into a Tail Recursion

length: LIST → INTEGER;
 	
axiom:: for all x: LIST ⇒
 	
isempty x → length x = 0,
¬ isempty x → length x = length(tail x) + 1;
	length: LIST → INTEGER;
	len: LIST→ INTEGER→ INTEGER;
	axiom:: for all x: LIST; r: INTEGER ⇒
	length x r = 0 + r,
	isempty x → len x r = 0 + r,
	¬ isempty x → len x r = len(tail x) (1+r);

Figure 4-10. Application of a Transformation to Function "Length"

The transformation scheme is proven correct once and for all (as will be all transformation schemes used in PROSPECTRA) and the only "proof obligation" is to show that the applicability conditions in the "such that" part of figure 4-9 are met (this is trivial and, therefore, not developed).

Notes:

a. blabla: A -> B -> c

denotes the signature of a function "blabla" having two parameters of respective types A and B, and returning a result of type C.

b. :: separates a static denotation from an axiom or a precondition pertaining to the denotated element

c. ¬ denotes the boolean operator "not".

d. blabla a b denotes a type C expression (considering the blabla signature above) resulting from the evaluation of a function blabla with the actual parameters a (of type A) and b (of type B).

The algorithm on the right part of figure 4-10 can be translated in Ada without any further semantic transformation. The result of such a transformation is displayed in figure 4-11. The next development phase for function "length", the "imperative implementation" involves two extra transformations:

a. Transformation of the tail recursion in LEN into an imperative loop (we do not detail this very classical transformation scheme).

b. Unfolding of LEN (substitution of the statement return LEN(X,0) in LENGTH by the expansion of the body of LEN considering its actual parameters (X,0)).

The result of the successive application of both these transformations is in figure 4-12.

A very powerful and specific feature of PROSPECTRA is the combined use of algebraic specifications and functionals. Functionals are higher order functions (i.e., functions as parameters and/or result functions). This enables PROSPECTRA to factorize the specification of a number of different operations having a similar structure (such operations are said to be homomorphic operations) and the transformations which are necessary to implement these operations. Carrying on with the same example of LISTS, one can define an "homomorphic

```
function LENGTH (X: LIST) return INTEGER is
   function LEN (X: LIST; R: INTEGER) return INTEGER is
   begin
      if isempty (X) then return R;
                     else return LEN (TAIL(X), R+1));
      endif;
   end LEN;
begin
   return LEN(X,0);
end LENGTH;
```

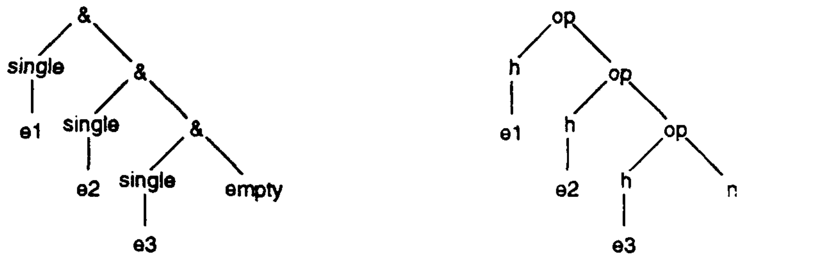
Figure 4-11. A Tail Recursive Ada Program for "Length"

```
function LENGTH (X: LIST) return INTEGER is
  V: LIST:=x; R: INTEGER:=0;
begin
  while not isempty(V) loop
    V:= TAIL(V);
    R:= R+1;
  end loop;
  return R;
end LENGTH;
```

Figure 4-12. The Final Implementation of "Length"

extension functional", called LIST_MONOID_HOM, which is an abstraction of the monoid structure of lists with the constructor "&" and the neutral element "empty" (a monoid is a type having an associative operator with a neutral element).

Figure 4-13 illustrates the principle of abstracting LIST_MONOID_HOM from the monoid (LIST, &, empty). Figure 4-14 gives the specification of LIST_MONOID_HOM which is nested in the specification of package LISTS together with different instantiations of LIST_MONOID_HOM. These instantiations are used to specify a number of useful operations implying to travel through a list (including "length' which is given in another (equivalent) specification).



Monoid (LIST,&,empty)

Homomorphic Extension Functional LIST_MONOID_HOM

Figure 4-13. Abstracting the Homomorphic Extension Functional LIST_MONOID_HOM From the Monoid (LIST,&,empty)

These operations (function and functionals) are as follows:

a. With the "AUTO_M" instantiation:

1. "Map" transforms a list by applying a function f: ITEM -> ITEM to each of its items.
2. "Filter" builds a sublist of the items of a list which satisfy a predicate p: ITEM -> BOOLEAN.
3. "Filt" has no use but to be an element of the specification of "filter".

b. With the "toINT-HOM" instantiation:

"length" gives the length of a list (the present specification is equivalent to the specification in figure 4-8).

c. With the "toBOOL_HOM" instantiation:

1. "Exist" returns true if one item at least of a list satisfies a predicate p: ITEM -> BOOLEAN.
2. "Forall" returns true if all the items of a list satisfy a predicate p: ITEM -> BOOLEAN.
3. "isElem" returns true if a list includes at least one element equal to a given value or element ("eq" is the equality operator used by "isElem").

The specification in figure 4-14 is much more compact than an equivalent specification on which the axioms pertaining to all the functions and functionals above would be separately developed without using the homomorphic extension functional "LIST_MONOID_HOM".

To eliminate the linear recursion which appears on the specification of "LIST_MONOID_HOM" in figure 4-14, we can apply three successive transformations the last of which is similar to the transformation in figure 4-9.

```
generic
  type ITEM is private;
  "<": ITEM→ ITEM→ BOOLEAN:: for all x, y, z: ITEM
            ⇒ ¬(x < y), x < y ∧ y < z - > x < z;
package LISTS is
  type LIST is private;
  empty: LIST;
  cons: ITEM → LIST→ LIST;
-----------------------------------------------------------
  "&": LIST→ LIST→ LIST;
  single: ITEM→ LIST;
axiom:: for all x, y, z: LIST ⇒
      x & empty =x, empty & x = x, (x & y) & z = x & (y & z);
-----------------------------------------------------------
  isempty: LIST → BOOLEAN;
  head: (X: LIST: ¬isempty x) → ITEM;
  tail: (x: LIST:: ¬isempty x) → LIST;
axiom:: for all e: ITEM; l: LIST⇒
      isempty empty = true, isempty (cons e l) = false,
      head (cons e l) = e, tail (cons e l) = l, (single e) & l = cons e l;
-----------------------------------------------------------
  "<=": LIST→ LIST→ BOOLEAN;
axiom:: for all l, l1, l2: LIST; x1, x2: ITEM
    ⇒ l <= l & l2, x1 < x2→ l & (single x1) & l1 <= l & (single x2) & l2;
-----------------------------------------------------------
  generic
   type M is private;
  package LIST_MONOID_HOM is
   Hom: (n: M)→ (op: M→ M→ M)→ (h: ITEM→ M)→ LIST→ M::
     for all x, y, z: M
     ⇒ Hom n op h empty = n,
         Hom n op h (x & y) = op (Hom n op h x) (Hom n op h y),
         Hom n op h (single e) = h e;
  end LIST_MONOID_Hom;
-----------------------------------------------------------
  package AUTO_M is new LIST_MONID_HOM (LIST); use AUTO_M;
  Map: (ITEM→ ITEM)→ LIST→ LIST;
  Filter: (ITEM→ BOOLEAN)→ LIST→ LIST;
  Filt: (ITEM→ BOOLEAN)→ ITEM→ LIST;
axiom:: for all f: ITEM → ITEM; p: ITEM → BOOLEAN; x: ITEM
          => Map f = Hom empty "&" (single f),
          Filter p = Hom empty "&" (filt p), p x→ Filt p x = single x, ¬ p x→ Filt p x = empty;
-----------------------------------------------------------
  package toINT_HOM is new LIST_MONOID_HOM (INTEGER); use toINT_HOM;
  length: LIST → INTEGER;
  one: ITEM→ INTEGER;
axiom:: for all x: ITEM => one x = 1, length = Hom 0 "+" one;
-----------------------------------------------------------
  package toBOOL_HOM is new LIST_MONOID_HOM (BOOLEAN); use toBOOL_HOM;
  Exist: (ITEM → BOOLEAN) → LIST→ BOOLEAN;
  Forall: (ITEM→ BOOLEAN)→ LIST→ BOOLEAN;
  eq: ITEM→ ITEM→ BOOLEAN;
  iselem: ITEM→ LIST→ BOOLEAN;
axiom:: for all x: ITEM; a, b: LIST =>
  Exist = hom false "or", Forall = Hom true "and",
  eq x y = not (x<y) and not (y<x), iseleme x = Exist (eq x);
-----------------------------------------------------------
end LISTS;
```

Figure 4-14. Extension of the Specification of LISTS With a Number of Operations Defined Using the Homomorphic Extension Functional LIST_MONOID_HOM

Figure 4-15 gives three sets of axioms, separated by horizontal bars, which result from the application of these successive transformations. On the last set of axioms, only a tail recursion remains on the auxiliary functional H2. An efficient implementation of LISTS should necessitate applying further, more classical, transformations which will not be detailed here since they are analogous to those already shown on function "length". The homomorphic extension function "LIST_MONOID_HOM" has a number of interesting properties which are reported in figure 4-16. Of course other homomorphic extension functionals can be similarly defined on other structures than lists (sets for example).

4.4.1.5 PROSPECTRA: Conclusions, Expectations and Hopes

Starting from several preexisting elements, the project PROSPECTRA provided significant advances in formal program development by transformations. These advances are mainly:

a. Program optimization using recursion removal transformation (this has been open for a long time but PROSPECTRA made some progress in this direction).

b. Formalization of transformation rules and of the program development process; as well as introduction of the capability of replaying a development (to be extended to the capability of modifying a replayed development).

One of the most significant PROSPECTRA advances in this area has been the combination of the possibilities offered by algebraic specifications and functional programming.

The works in progress and/or planned until the end of the project (March 1990) are:

a. Incorporating new elements allowing the development of concurrent programs (only sequential program development has been considered so far in the project).

```
generic
   type ITEM is private;
   "<": ITEM→ ITEM→ BOOLEAN::
        for all x, y, z:ITEM
        ⇒ ¬(x < y), x < y ∧ y < z − > x < z;
package LIST is
—idem Figure 5
-------------------------------------------------------------
   generic
    type M is private;
   package LIST_MONOID_HOM is
    Hom: (n: M)→ (op: M→ M→ M)→ (h: ITEM→ M)→ LIST→ M ::
      for all x, y, z:  M⇒ op x n = x,
      op (op x y) z = op x (op y z);
axiom for all n:M; op: M→ M→ M; h: ITEM→ M; e: ITEM; y, z:LIST;
     r: M⇒
           Hom n op h empty = n, -linearize by substitution
           Hom n op h ((single e) & y) = op (h e) (Hom n op h y),
-------------------------------------------------------------
   isempty z → Hom n op z = n' -leminate constructors
   ¬ isempty z → Hom n op h z = op (h (head z))(Hom n op h (tail z)),
-------------------------------------------------------------
                  Hom n op h z = H2 n op h n z, -eliminate recursion
   isempty z → H2 n op h r z = r,
   ¬ isempty z → H2 n op h r z = H2 n op h (op (h (head z)) r) (tail z);
-------------------------------------------------------------
   end LIST_MONOID_HOM;
-------------------------------------------------------------
-idem Figure 5
end LISTS;
```

Figure 4-15. Result Obtained After Removing the Linear Recursion on LIST_MONOID_HOM.

```
generic
  type ITEM is private;
  "<": ITEM→ ITEM→ BOOLEAN::
      for all x, y, z:ITEM
      ⇒ ¬(x < y), x < y ∧ y < z − > x < z;
package LIST is
—idem Figure 5
-----------------------------------------------------------------
  generic
   type M is private;
  package LIST_MONOID_HOM is -idem Figure 5
-----------------------------------------------------------------
  package AUTO_M is new LIST_MONOID_HOM (LIST); use AUTO_M;
–idem Figure 5
axiom for all n: M; op: M→ M → M; h: ITEM→ M; x, y: LIST;
          f, g: ITEM → ITEM; p, q: ITEM→ BOOLEAN⇒
(1) (Hom n op h)•(Map g) = Hom n op (h•g),
(2) (Map f)• (Map g) = Map (f •g),
(3) Map f (x & y) = (Map f x) & (Map f y),
(4) Filter p (x & y) = (Filter p x) & (Filter p y),
(5) (Filter p)• (Filter q) = (Filter q)•(Filter p),
(6) (Filter p)• (Map f) = (Map f)•(Filter (p • f)),
(7) (Filter p)• (Map f) = Hom empty "&" ((Filt p)•f),
-----------------------------------------------------------------
  package toINT_HOM is new LIST_MONOID_HOM (INTEGER); use toINT_HOM;
–idem Figure 5
-----------------------------------------------------------------
  package toBOOL_HOM is new LIST_MONOID_HOM (BOOLEAN); use toBOOL_HOM;
–idem Figure 5
-----------------------------------------------------------------
end LISTS;
```

Figure 4-16. Some Properties of LIST_MONOID_HOM.

b. Replacing the present implementation under the Cornell synthesizer by a more efficient one.

c. Adding to transformations applicability conditions depending on the context.

d. Extending the set of transformations and making it more modular.

e. Preparing a final report on the project to be published by SPRINGER-VERLAG (reference 4-55).

Beyond the end of PROSPECTRA further works are foreseen on:

a. Extending the system to make it applicable to large software development projects (not presently the case).

b. Developing new application domain specific transformation rules.

c. Developing training courses which could give people a working knowledge of formal methods.

4.4.2 TOOLUSE

TOOLUSE is an advanced support environment for method-driven developments. It is also the continuation of the SPRAC project.

4.4.2.1 General Overview

The TOOLUSE general objectives (ref. 4-28) are to provide active assistance in the various activities of software development through the formalization and support of development methods. This formalization is done through DEVA, a language used to express the design decisions related to methods as well as a specification language.

This section deals with the linguistic framework used to formalize methods and presents examples to illustrate DEVA. Indeed, transformational techniques are not restricted to specifications and/or code. They should be applied to program developments as well as extending reusability techniques instead of being restricted to reuse of components. More comments are in reference 4-29.

4.4.2.2 DEVA Program Development

Let us consider Jackson's method which is strongly based on transformation. In reference 4-30 a formalization of all the JSP/JPD transformations has been introduced. The way such transformations can be automatized is schematized in figure 4-17. The overall structure of the example, described in figure 4-17, is:

a. Theory-JPD introduces the basic formal transformational rules about regular expressions and about correspondences among these expressions.
b. Dev-Theory-JPD gives the development knowledge related to the ISP method.
c. Application introduces the current problem specification.
d. Application-Development corresponds to the current development that starts from the specification and ends with the concrete program.

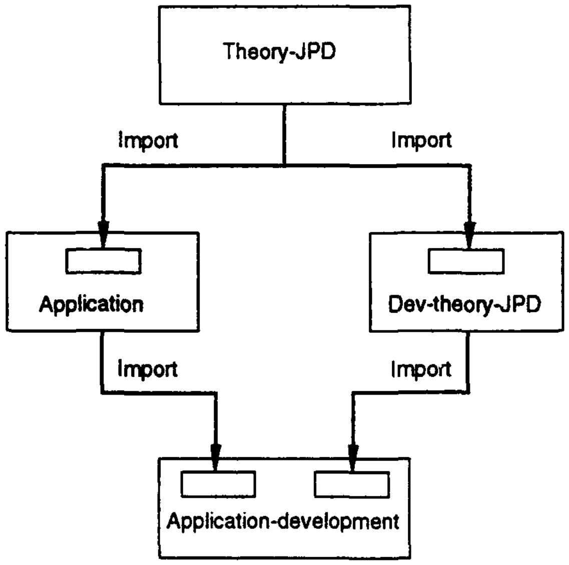


Figure 4-17. JPD With DEVA

4.4.2.2.1 A Small Example

In the following paragraphs, we present a small example developed with an available prototype for DEVA. Due to implementation reasons, notations differ slightly from those presented above.

4.4.2.2.2 The Problem To Be Solved: Informal Requirements

We are now considering a small problem presented by M. Jackson in reference 4-31 and examplified in reference 4-32. Very briefly, the problem can be stated as: *Let us consider daily statistical data on rain in a given area. How much rain has fallen each week?* The input data are recorded in a file and organized as follows:

a. Daily records (Drec) beginning with the symbol D and followed by an identifier of the day (Dno) and the amount of rain for the considered day (Damount).
b. Weekly records (Wrec) composed of the symbol W and an identifier of the week (Wno).

All the records are ordered according to the date. The output data must be recorded as weekly records in which the amount of rain for each week has been computed. In other words, we would like to find the relationship between Drec and Wrec and to derive the value of Wamount as a function of Dno, Damount.

4.4.2.2.3 Problem Formal Description

The problem statement is formalized in DEVA, below, using the Theory-JPD introduced before.

```
part RAIN:=
    part begin
      import part Theory-JPD
    ; input, output, Wgroup, Wrec, Drec, Wno  :  exp
    ; Wamount, Henreg, W, D, Dno, Damount  :  exp
    ; I1  :  input    =  (*Wgroup)
    ; I2  :  Wgroup  =  (Wrec . (*Drec))
    ; I3  :  Wrec     =  (W . Wno)
    ; I4  :  Drec     =  (D . (Dno . Damount))
    ; O1 :  output   =  (*Henreg)
    ; O2 :  Henreg  =  (Wno . Wamount)
    end
```

A few comments:

a. To reuse Jackson's definitions and rules, we need to import part Theory-JPD
b. Input is the input stream. It is a list of Wgroup.
c. Wgroup is the concatenation of Wrec (week record) and a list of Drec (day record).
d. Wrec is defined as W followed by Wno.
e. Drec is the sequence of D and Dno (day number) and Damount
f. Output is the output stream. It is a list of Henreg.
g. Henreg is defined as Wno followed by Wamount.

At this level, it is important to notice that we use declarations of equality and not DEVA definitions as for instance input:= (*Wgroup). The equality is actually an operator belonging to the object language and, as such, it can be directly manipulated.

The definition of part RAIN is not complete. Indeed, the definitions we have given are all of type exp. To match in full details Jackson's problem, different types and constraints must be defined. For instance, Wno represents a week number, and consequently should be of type *integer-less-than-53*. Dno the day number should be typed by *integer-greater-than-0-and-less-than-8*. Damount is an amount expressed in conventional unit (m or mm).

4.4.2.2.4 First Step: How Transformations Are Proved Applicable

Step 1 starts from the hypothesis that the input stream input and the output stream output correspond. We would like to deduce that a good transformation is obtained by the substitution of the input (respectively the output) by its value (declarations I1 and O1 in part Theory-JPD). This can be achieved in a proved manner with the following DEVA text:

```
from part begin
           import part RAIN
        ;  e0  :=   input
        ;  e1  :=   (*Wgroup)
        ;  s0  :=   output
        ;  s1  :=   (*Henreg)
        ;  hypothesis:  e0 -> s0
        ;  conclusion1  := e1 -> s1
        ;  thesis1   :=    [hypothesis |- conclusion1]
        ;  proof1    :=    [hypothesis
                           |-
                           ax1 (e0, s0, e1, s1)
                           (hypothesis,I1,O1)]
        end
infer
        proof1 .: thesis1
end
```

Some comments:

a. A development (or proof) is written according to the general schema:
   FROM context INFER proof .: thesis.
   Context is the context in which the development is done. *Proof* and *thesis* in our example are elements of the defined context.
b. We need to import the RAIN context (which imports Theory-JPD).
c. We define intermediate variables e0, e1, s0, s1 seen only as abbreviations to simplify the formulas.
d. Hypothesis is defined as the establishment of the correspondence between the input stream input and the output stream output.
e. Conclusion 1 states that there is a correspondence between (*Wgroup) and (*Henreg).
f. Thesis 1 says that from the hypothesis we can derive the conclusion.

4.4.2.2.5 Second Step: Applying Other Transformations

Step 2 is the same except that we start as hypothesis from the conclusion established in step 1, and then we derive a new conclusion which is a correspondence between more refined input e2 and output s2. But the overall schema is the same.

4.4.2.2.6 Combining Step 1 and Step 2

Combining several development steps is possible through DEVA text application as follows:

```
from part begin
          import part RAIN
        ; e0  :=  input
        ; e1  :=  (*Wgroup)
        ; e2  :=  (*(Wrec . (*Drec)))
        ; s0  :=  output
        ; s1  :=  (*Henreg)
        ; s2  :=  (*(Wno . Wamount))
        ; hypothesis : e0 ⇒ s0
        ; conclusion2 : e2 ⇒ s2
        ; thesis2 := [ hypothesis | - conclusion2]
        ; step1 := [ h0: e0 ⇒ s0
                    | -
                      ax1 (e0,s0,e1,s1,h0,I1,O1)
                                    .: e1 ⇒ s1
        ; step2 := [ h: e1 ⇒ s1
                    | -
                      ax2  (Wgroup, Henreg,
                           (Wrec . (*Drec)),
                           (Wno . Wamount)),
                           h,I2,O2) .: e2 ⇒ s2 ]
        ; proof2 :=  step2 (step1(hypothesis))
                     { which are text applications }
          end
infer
          proof2 .: thesis2
end
```

A few comments:

a. At step 1, it is certified that, from the hypothesis, we can derive an intermediate result establishing that *e1 corresponds to s1*.

b. At step 2, it is certified that, from *e1 corresponds to s1*, we can derive the new conclusion *e2 corresponds to s2*.

c. The proof is the composition of step 1 and step 2 applied to the hypothesis.

4.4.2.2.7 Improvements and Conclusions

DEVA is a higher order language based on a typed λ-calculus. It constitutes a uniform framework allowing expression of different kinds of objects such as developments, steps of developments, or programs (i.e. results of developments). For example:

a. A development is expressed as a λ-term,

b. A program is also expressed as a λ-term and, using the *propositions as types analogy*, is defined as the type of the λ-term expressing its development.

What was done for the derivation of actual developments can be done on the developments considered as objects. This capability is mainly studied from the point of view of reusing techniques in the REPLAY project (project 1651 in ESPRIT).

The particular contribution of DEVA (compared to some deductive environments such as LCF, where tactics are expressed in a colanguage or a metalanguage is to integrate control facilities into the language itself. Some basic control operators have been introduced for the definition of tactics which are then considered as first class citizens and can be used as transformation rules.

4.5 TRANSFORMATIONS OF CONCURRENT PROGRAMS

4.5.1 General Outline

The transformational approach to the development or analysis of concurrent programs (or more generally concurrent systems) has mainly been worked out with the PETRI nets model. PETRI nets transformations are generally based on net morphisms, a specific class of the already presented FOLDING/UNFOLDING transformations. Some transformations are aimed at helping to prove properties of an already specified system (ref. 4-33). Such transformations do not preserve the general semantics of the system but produce a simplified model preserving some interesting properties of the original system (boundedness, deadlock freeness, liveness), thus helping the analysis of these properties in the system.

Other transformations are more directly involved in a system development cycle such as:

a. Transformations starting from the informal requirements and progressively adding structure and formal semantics to the system (ref. 4-34).

b. Transformations starting from formal specifications of an object or abstract data type to implement it in terms of structures of a high-level programming language such as Ada (ref. 4-35).

The latter category of transformations, which concerns software generation more directly, will be developed in the following paragraphs.

4.5.2 An Introduction to the Specification of Abstract Data Types for Concurrent Programs Using Predicate Transition Nets and to a Transformational Approach for Their Implementation in Ada

We present an extension of the transformational approach to programming (the subject of major works for more than 10 years in the domain of sequential programs) to the domain of concurrent programs.

This approach is based on the use of predicate-transition nets (or Pr-T nets), a high-level form of PETRI nets. Section 4.5.3 presents a model-oriented style of specification in which programs are specified as collections of asynchronous processes using common objects which export the necessary synchronization and communication operations. Section 4.5.4 deals with the specification of abstract data types defining classes of such objects. A set of specification schemes for such abstract data types is presented. It is rather high level in the sense that it does not constrain its users to pass through a predetermined style of synchronization/ communication; it leaves them free to stay close to their problem domain. Section 4.5.5 presents the transformational approach used to implement such specifications into executable Ada code. A basic transformation, used for all implementations, together with a general transformation strategy are presented. Then an example of implementation of an abstract data type is demonstrated. This example demonstrates the technique:

a. The problem is complex enough to be almost untreatable by nonformal techniques (such as pseudo-code).
b. The transformations used for this implementation constitute a representative sample of our overall transformation set.

Finally, section 4.5.6 presents the current status of this work.

4.5.3 Specification of Concurrent Programs Modelled as a Collection of Asynchronous Processes

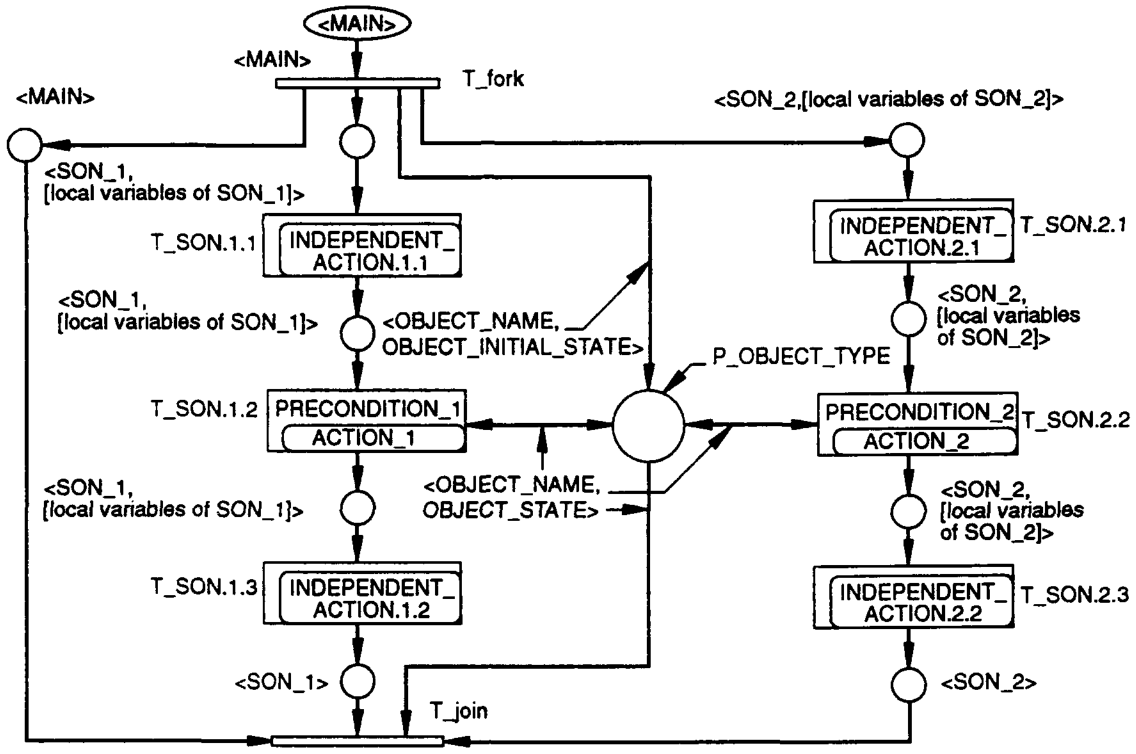
Concurrent programs are modelled as a composition of asynchronous processes which proceed independently except at points where they synchronize (synchronization points). Note that the "asynchronous" qualifier qualifies the processes and not their synchronization mode which can be synchronous (such as rendezvous) as well as asynchronous (such as mailboxes). This general definition is adopted by a large majority of models (such as CSP [Hoa78], [Hoa85], CCS [Mil80], [Mil84} and its derivatives like SMOLCS [AG85] and programming languages (such as Ada [Ada87], LTR [LTR85], RTL [Bar76], PEARL [PEA80]. One essentially distinguishes three kinds of synchronization points:

a. Fork: creation of asynchronous processes.
b. Join: wait on the termination of asynchronous processes.
c. Synchronization through the use of common objects exporting synchronization/communication operations.

Other kinds of synchronization points may owe their existence to the necessity of handling signals coming from the external world or of abruptly stopping a process, but they are outside the scope of the present study. On the contrary, some models may admit static processes only, which are all created at the program start point, and, therefore, have no fork synchronization points and often no join synchronization point.

As well as local objects, the common objects used for interprocesses synchronization/communication will generally have their specification grouped into a smaller number of abstract data types; each abstract data type factorizing the properties which are common to a class of objects. An example of a specification of concurrent program is in figure 4-18. As all our specifications, this one has two parts:

a. A textual part giving the syntax, the static semantics and the purely sequential part of the dynamic semantics. As far as possible, we use the language Ada to describe the static part of these specifications. (This tends to minimize the number of different formalisms we use, since our programs are finally implemented in Ada.)
b. A predicate-transition net (ref. 4-36), or Pr-T net, which expresses the concurrency properties of the program by modelling it as a collection of asynchronous processes.



<MAIN>
<MAIN>
T_fork
<MAIN>
<SON_2,[local variables of SON_2]>
<SON_1, [local variables of SON_1]>
T_SON.1.1
INDEPENDENT_ ACTION.1.1
INDEPENDENT_ ACTION.2.1
T_SON.2.1
<SON_1, [local variables of SON_1]>
<OBJECT_NAME, OBJECT_INITIAL_STATE>
<SON_2, [local variables of SON_2]>
P_OBJECT_TYPE
T_SON.1.2
PRECONDITION_1 ACTION_1
PRECONDITION_2 ACTION_2
T_SON.2.2
<SON_1, [local variables of SON_1]>
<OBJECT_NAME, OBJECT_STATE>
<SON_2, [local variables of SON_2]>
T_SON.1.3
INDEPENDENT_ ACTION.1.2
INDEPENDENT_ ACTION.2.2
T_SON.2.3
<SON_1>
<SON_2>
T_join

Where:

```
OBJECT_NAME : OBJECTS.OBJECT_IDENTIFIER;
OBJECT_STATE : OBJECTS.OBJECT_TYPE;
package OBJECTS is
  type OBJECT_TYPE is...;
  OBJECT_INITIAL_STATE : constant OBJECT_TYPE :=...;
  type OBJECT_IDENTIFIER is limited private;
  procedure OPERATION_1 (OBJECT_NAME : in out OBJECT_IDENTIFIER;...);
  --PRECONDITION : {PRECONDITION_1};waiting precondition
  --POSTCONDITION : {POSTCONDITION_1};
  procedure OPERATION_2 (OBJECT_NAME : in out OBJECT_IDENTIFIER;...);
  --PRECONDITION : {PRECONDITION_2};waiting precondition
  --POSTCONDITION : {POSTCONDITION_2};
private
  --not a part of the specification
end OBJECTS;
------------------------------------------------------------------------
with OBJECTS;
procedure MAIN;
------------------------------------------------------------------------
```

Figure 4-18. Example of a Specification of Concurrent Program

Examining the "a" part of the specification in figure 4-18, we see that:

a. The program

is called MAIN ("procedure MAIN"),

uses an abstract data type OBJECTS ("with OBJECTS"),

has an empty external interface (procedure MAIN has no parameter, no precondition, and no postcondition);

b. The abstract data type OBJECTS defines a class of objects which incorporates a logical structure defined by the type OBJECT_TYPE and initialized with the value OBJECT_INITIAL_STATE, each object of which is identified by an OBJECT_IDENTIFIER type variable, which exports two operations OPERATION_1 and OPERATION_2.

c. Each of the operations exported by the abstract data type OBJECTS is defined by:
   1. Its parameters.
   2. A precondition.
   3. A post condition.

   The precondition may have two parts:
   1. An absolute precondition which must hold when the operation is invoked.
   2. A waiting precondition: the process which invokes the operation waits until this precondition holds.

In the specific case of operations, OPERATION_1 and OPERATION_2, there is only a waiting precondition: this will be true for almost all the examples and schemes of specifications presented in this paper.

Examining the "b" part of the specification in figure 4-18, we see that:

a. Each process is modelled by a token with a tuple structure; the first element is the name of the process and the other elements are variables which are local to the process (<MAIN>, <SON_1, ...>, <SON_2, ...>).
b. The process MAIN is the only element which is created at program start (the upper place of figure 4-1 net is the only place to have an initial marking, the token <MAIN>).
c. The behavior of each process (or type of process in a more general case) is defined by a state machine composed of:

   transitions T_fork and T_join for process MAIN,
   transitions T_SON.1.1, T_SON.1.2, T_SON.1.3 for process SON_1,
   transitions T_SON.2.1, T_SON.2.2, T_SON.2.3 for process SON_2.

d. The objects of abstract data type OBJECTS are each modelized by a token normally staying (when the object is available) on the place P_OBJECT_TYPE.
   An object token has two elements:
   1. The object identifier, belonging to the type OBJECT_IDENTIFIER.
   2. The logic state of the object, belonging to the type OBJECT_TYPE.
e. The program behavior is as follows:
   1. Process MAIN creates processes SON_1 and SON_2 and object OBJECT_NAME of the abstract data type objects (firing transition T_fork) and then waits until processes SON_1 and SON_2 terminate (or transition T_join).
   2. Processes SON_1 and SON_2 execute independent actions (firing transitions T_SON.1.1 and T_SON.2.1), have some interaction (synchronization and/or communication) through their respective calls to operations OPERATION_1 (transition T_SON.1.2) and OPERATION_2 (transition T_SON.2.2) on object OBJECT_NAME, and then resume independent execution (firing transitions T_SON.1.3 and T_SON.2.3).

3. When process SON_1 has fired transition T_SON.1.1, it fires transition T_SON.1.2 (executes OPERATION_1 on object OBJECT_NAME) if PRECONDITION_1 holds, else it must wait until PRECONDITION_1 holds which can only happen as a result of process SON_2 having executed OPERATION_2.

   The execution of OPERATION_1 has the effect of accomplishing ACTION_1 which results in having POSTCONDITION_1 hold. Similar statements can be made about process SON_2 and transition T_SON.2.2.

   In this example, we see the three kinds of synchronization points:

1. Fork: transition T_Fork,
2. Join: transition T_join,
3. Synchronization through use of a common object (OBJECT_NAME) between processes SON_1 and SON_2 (transitions T_SON.1.2 and T_SON.2.2).

f. The rest of this section is centered on specification of abstract data types (represented in the figure 4-18 example by package OBJECTS and the subnet formed by transitions T_SON.1.2, T_SON.2.2 and by place P_OBJECT_TYPE) and on the implementation of such specifications in Ada using a transformational approach.

g. The Ada language (ref. 4-37) was chosen as a target of implementation essentially because it is the most portable intermediate target that we presently have.

Our general approach for specification and implementation by transformations will still be valid with another target, except that the transformations which are presented in section 4.5.4 are target dependent (Ada, in the present case).

4.5.4 Abstract Data Type Specification

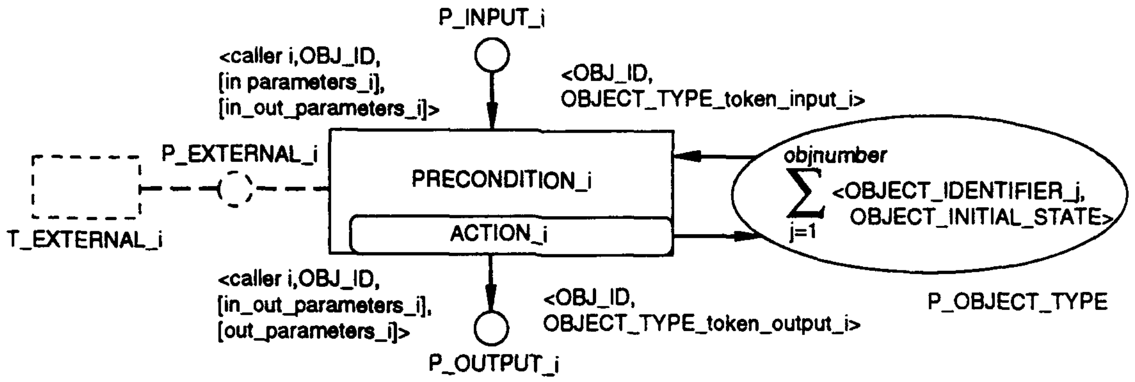
4.5.4.1 A General Scheme

Figure 4-19 presents a rather general scheme of an abstract data type specification. We mainly see in figure 4-19:

a. The place P_OBJECT_TYPE of the abstract data type OBJECTS with objnumber static objects of the type (which does not preclude a possible creation of other dynamic objects).

b. An elementary operation scheme OPERATION_i with a single transition T_i accomplishing an indivisible action on an object (the action is indivisible because transition T_i removes the object token from place P_OBJECT_TYPE during its fire).

The action ACTION_i accomplished by firing transition T_i results in having the postcondition POSTCONDITION_i hold. POSTCONDITION_i may imply that specific values are assigned to the output parameters and to the logic state of the object (this means of course that OBJECT_TYPE_token_output_i may be different from OJBECT_TYPE_token_input_i).


P_INPUT_i
<caller i,OBJ_ID, [in parameters_i], [in_out_parameters_i]>
<OBJ_ID, OBJECT_TYPE_token_input_i>
P_EXTERNAL_i
T_EXTERNAL_i
PRECONDITION_i
ACTION_i
objnumber
$\sum_{j=1}$ <OBJECT_IDENTIFIER_j, OBJECT_INITIAL_STATE>
<caller i,OBJ_ID, [in_out_parameters_i], [out_parameters_i]>
<OBJ_ID, OBJECT_TYPE_token_output_i>
P_OBJECT_TYPE
P_OUTPUT_i

Note:
The interaction of T_i with the "external world" (P_EXTERNAL_i,T_EXTERNAL_i) must not involve any waiting condition for T_i.

```
package OBJECTS is
  type OBJECT_TYPE is...;
  OBJECT_INITIAL_STATE : constant OBJECT_TYPE :=...;
  type OBJECT_IDENTIFIER is limited private;
  ...
  ...
  procedure OPERATION_i(OBJ_ID : in out OBJECT_IDENTIFIER;
        [formal_in_parameters_i];
        [formal_in_out_parameters_i];
        [formal_out_parameters_i];
  --PRECONDITION : {PRECONDITION_i};
  --POSTCONDITION :{POSTCONDITION_i};
  --holds after ACTION_i has been performed;
  ...
  ...
private
  --not a part of the specification
end OBJECTS;
```

Figure 4-19. The General Specification Scheme for an Abstract Data Type

The action ACTION_i may also have a nonsequential interaction with the external world (which is modelled in figure 4-19 by macroplace P_EXTERNAL_i and macrotransition T_EXTERNAL_i; these nodes are in dotted lines to indicate that they are not fully detailed in this net), but this interaction may not imply any waiting condition during ACTION_i execution (this is justified by the elementary characteristic of T_i which preempts the object token during its fire).

The possible variations to this general scheme are as follows:

a. An elementary operation may involve several objects simultaneously, or one object which is selectively chosen from several instead of being designated.
b. All the parameter passing modes (in, in out, out) are not present in every elementary operation.
c. An elementary operation may or may not have a nonwaiting interaction with the external world.
d. The wait on the waiting preconditions may be limited by a time out.
e. The precondition of an elementary operation may have an absolute and a waiting part.
f. When an absolute precondition is present in an elementary operation, two alternative behaviors may be given when it does not hold on a call of the operation:
   1. Raise an exception (the declaration of which will preferably either be given locally to the abstract data type specification or be directly inherited.
   2. Consider the subsequent behavior of the program as erroneous (i.e., random).

g. An abstract data type may export "nonelementary" operations which are a combination of elementary operations (an example of such an abstract data type will be seen later).

h. Add the possibility of creating and suppressing dynamic objects.

i. An abstract data type may be made generic; the Ada package which is a part of its specification is then a generic package.

j. An abstract data type specification may use another abstract data type specification, the Ada package specification then contains a "with" clause pointing on the other abstract data type package specification.

An abstract data type specification may also inherit from another abstract data type specification; the denotation of this fact in terms of Ada package specifications is more complex and will not be discussed here (details on this point and some possible extensions of the Ada language to better cover it can be found in references 4-38 through 4-40.

4.5.4.2 A Library of Elementary Operation Specification Schemes

Some of the variations listed above lead to substantial changes in the nature of program transformations that are used to implement an abstract data type into Ada language structures. Such variations must be distinguished by creating corresponding specification schemes which are more dedicated than the general specification scheme seen in figure 4-19.

We have a library of eight elementary operation specification schemes:

a. Four schemes correspond to different categories of waiting preconditions:
   1. The operation is on a designated object and the precondition depends only on the object structure.
   2. The operation is on a designated object and the precondition depends also on some input parameters.
   3. Selecting precondition case: The operation is on the first available object which makes the precondition hold.
   4. The operation involves several designated objects.

b. The schemes 5, 6, 7, 8 are respectively derived from the schemes 1, 2, 3, 4 by adding a time out on the waiting precondition. These eight schemes form a very general specification tool which allows one to specify any abstract data type exporting synchronization/communication properties.

The fact that this set of specification schemes is so general comes from:

a. The eight schemes covering all the possible cases of waiting precondition.

b. The absence of any predetermined scheme for the action part of the elementary transitions.

We will see in the next section that the abstract data types from this specification scheme set resemble a category of Ada task types called pure server tasks and, in some specific cases, may be directly implemented into such task types, but are actually much more general. The user, at the specification level that was the subject of this section, is absolutely not constrained to use the Ada tasking idioms nor any comparable low-level tasking idioms. This is an essential feature which distinguishes our methodology from most of those used in, or proposed for, industrial applications.

4.5.5 Implementation in Ada of Abstract Data Types Built with our Specification Schemes

4.5.5.1 Introduction

The aim is to generate correct, efficient, and portable programs from specifications built as described in the previous sections. The main advantages of this specification technique are:

a. To stay at a sufficiently abstract level (we do not use any predetermined low-level synchronization model) which enables us to easily specify any system for which an asynchronous process model can reasonably be given.

b. To be executable through net simulation.

c. To allow proofs of properties using techniques from references 4-36 and 4-41, such as:

   1. Computing Pr-T net invariants, some of which support the proof of useful properties.
   2. Analysis/reachable marking or coverage trees for the Pr-T nets.
   3. Reducing Pr-T nets.

After being able to specify a system and analyze its properties by proofs or simulations, the next step is to implement it into an efficient and portable program which preserves the properties of the specification (correct implementation). The approach to implement specifications into Ada programs uses successive program transformations.

The concurrent program transformations which are used here have, at their highest level, as input schemes the specification schemes which were presented in section 4.5.4. Each transformation has an output scheme which is closer to the Ada Language structures than its input scheme, so that, for the lowest level transformations, we have output schemes which contain only Ada language structures. These program transformations were obtained by generalizing previous works (refs. 4-42 and 4-43) which presented some examples of implementation of abstract data types defined using our specification schemes. They use a Pr-T net definition of the Ada tasking semantics which were presented in (ref. 4-44).

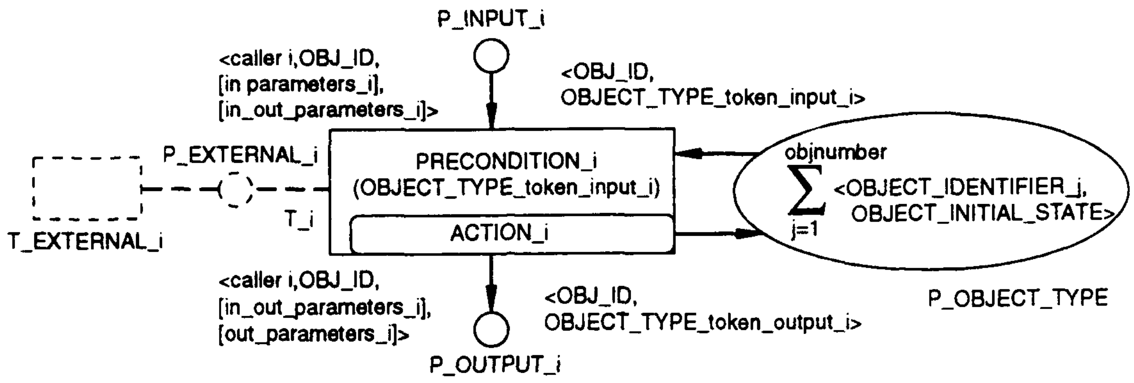
4.5.5.2 A Basic Transformation - Transformation Strategy

The basic transformation allows a direct implementation in Ada of abstract data types built exclusively with operations corresponding to the specification schemes 1 and 5 (such abstract data types may export elementary operations mapping scheme 1 or 5 or nonelementary operations built from a sequential composition operations mapping scheme 1 or 5). This basic transformation is presented in figure 4-20 (input scheme) and figure 4-21 (output scheme). The properties defined for abstract data types, built exclusively with elementary operations mapping specification scheme 1 (input scheme of the basic transformation in figure 4-20), are the following:

a. The elementary operations on a given object are divisible, since their transition (T_i) removes the object token from place P_OBJECT_TYPE during their firing. Formally, we could equivalently state that all firing sequences on such transitions are purely sequential:

   Given the definition of a firing sequence by the two following operators:

   1. $\alpha\beta$: Transition α fires and then transition β fires (sequential composition of α and β.
   2. $\alpha // \beta$: Transitions α and β fire concurrently (parallel composition of α and β.



Note:
The interaction of T_i with the "external world" (P_EXTERNAL_i,T_EXTERNAL_i)
must not involve any waiting condition for T_i.

```
package OBJECTS is
  type OBJECT_TYPE is...:
  OBJECT_INITIAL_STATE : constant OBJECT_TYPE :=...;
  type OBJECT_IDENTIFIER is limited private;
  ...
  ...
  procedure OPERATION_i(OBJ_ID : in out OBJECT_IDENTIFIER;
        [formal_in_parameters_i];
        [formal_in_out_parameters_i];
        [formal_out_parameters_i];
  --PRECONDITION : {PRECONDITION_i};
  --may only depend from the structure of the object designated by the OBJ_ID
  --POSTCONDITION :{POSTCONDITION_i};
  --holds after ACTION_i has been performed;
  ...
  ...
private
  --not a part of the specification
end OBJECTS;
```

Figure 4-20. The Basic Transformation Input Scheme

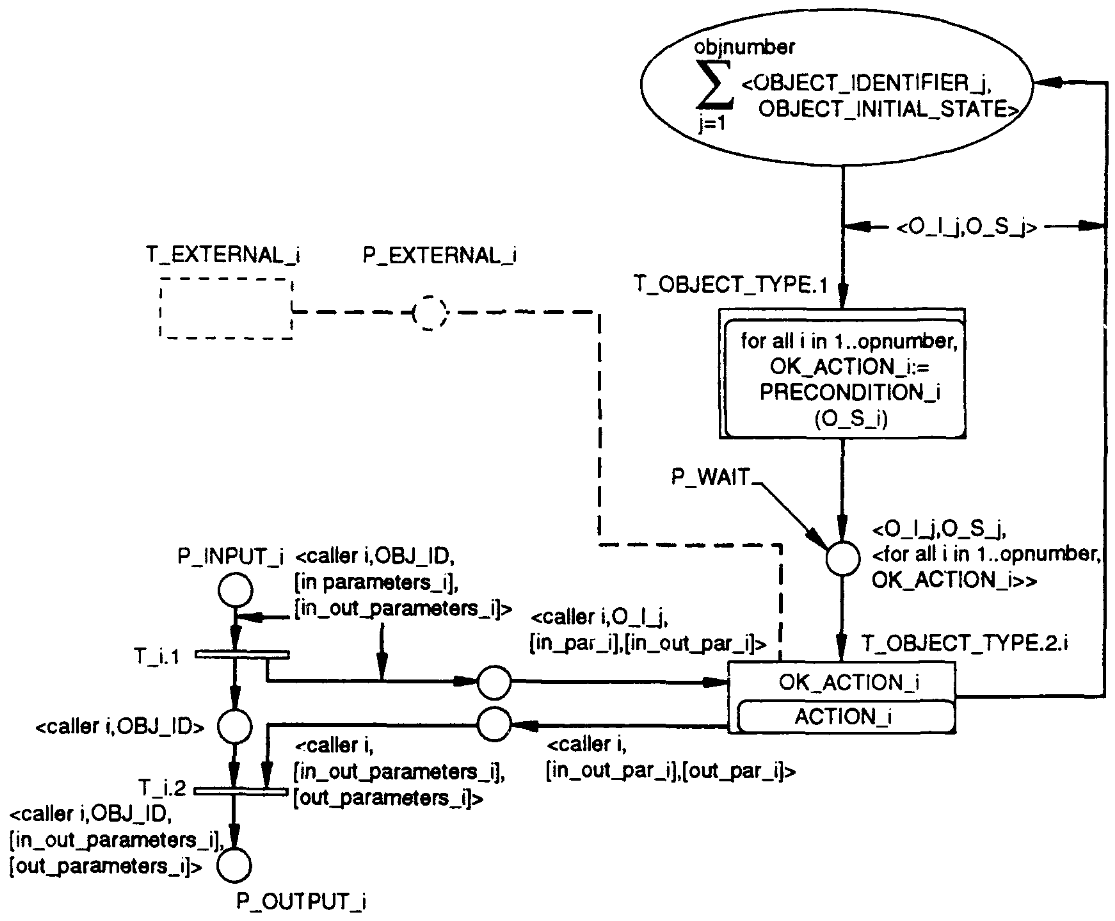
Let Σ OBJ_ID be the set of all possible firing sequences of transitions T_i for the object OBJ_ID, let T_i/OBJ_ID denote the firing of transition T_i for the object OBJ_ID and (T_i)*/OBJ_ID be the set of all possible words elaborated on the alphabet T_i/OBJ_ID, ..., T_1/OBJ_ID, ..., T_opnumber / OBJ_ID, where 1.. opnumber is the interval of indexes of all the elementary operations.

Then, Σ OBJ_ID=(T_i)*/OBJ_ID.

Note: This net is semantically equivalent to an Ada program scheme (T_i.1, T_i.2 matching an entry call; T_OBJECT_TYPE.1 matching the first part of a selective wait (the computation of the guards) and T_OBJECT_TYPE.2 matching an "accept" branch of this selective wait).

b. T_i/OBJ_ID is firable if:

1. A caller (token) is present on P_INPUT_i for the object OBJ_ID.
2. The object OBJ_ID token is on place P_OBJECT_TYPE (equivalently, for no j ≠ i T_j/OBJ_ID is firing).
3. PRECONDITION_i (OBJ_ID).



objnumber
∑ <OBJECT_IDENTIFIER_j, OBJECT_INITIAL_STATE>
j=1
<O_I_j,O_S_j>
T_EXTERNAL_i
P_EXTERNAL_i
T_OBJECT_TYPE.1
for all i in 1..opnumber, OK_ACTION_i:= PRECONDITION_i (O_S_i)
P_WAIT
<O_I_j,O_S_j, <for all i in 1..opnumber, OK_ACTION_i>>
P_INPUT_i <caller i,OBJ_ID, [in parameters_i], [in_out_parameters_i]>
<caller i,O_I_j, [in_par_i],[in_out_par_i]>
T_OBJECT_TYPE.2.i
T_i.1
OK_ACTION_i
ACTION_i
<caller i,OBJ_ID>
<caller i, [in_out_parameters_i], [out_parameters_i]>
<caller i, [in_out_par_i],[out_par_i]>
T_i.2
<caller i,OBJ_ID, [in_out_parameters_i], [out_parameters_i]>
P_OUTPUT_i

Figure 4-21. The Basic Transformation Output Scheme

(T_i/OBJ_ID) is firable) does not depend on the fields in_parameters_i and in_out_parameters_i of the token on P_INPUT_i.

c. The firing of T_i/OBJ_ID terminates in a finite amount of time provided ACTION_i does not loop (due to the absence of waiting condition within ACTION_i).

The replacement of some elementary operations mapping specification scheme 1 by elementary operations mapping specification scheme 5 would not change these properties. Equivalent properties to a, b, c above can be established on the output scheme of the basic transformation (fig. 4-21) by:

a. Substituting transition T_OBJECT_TYPE.2 on the output scheme to transition T_i on the input scheme.

b. Noticing that the only purpose of T_i.1 and T_i.2 on the output scheme is the decoupling of the caller processes from the object handler processes; transitions T_i.1 and T_i.2 do not introduce further waiting conditions or actions.

c. Noticing that T_OBJECT_TYPE.1 is a purely sequential computation (it has no precondition and no interaction with the outside world).

The transformation strategy will always aim at reaching a point where the basic transformation is applicable to finally get the Ada implementation. Some local transformations will then be necessary in some cases to optimize this Ada implementation. A set of transformations has been developed allowing successful application of this strategy to any abstract data type the specification of which is built from the eight specification schemes presented in section 4.5.4.2.

```
generic
  type ADDRESS is (< >);
package ADDRESSES is
  type ADDRESS_LIST is limited private;
--This package provides no protection for use by concurrent tasks.
--An ADDRESS_LIST is a list of<ADDRESS,POSITIVE> tuples.
--It is initially empty.
  NON_LISTED_ADDRESS:exception;
  procedure LIST_INPUT (AD : ADDRESS;
                        LIST : in out ADDRESS_LIST);
  --POSTCONDITION : if no<AD,N : POSITIVE> tuple is present
  --                  then the <AD,1> tuple is added to the LIST;
  --                  else N := N+1;
  --                end if;
  procedure LIST_OUTPUT (AD : ADDRESS;
                         LIST : in out ADDRESS_LIST);
  --PRECONDITION :
  --   a <AD,N : POSITIVE> tuple is in the LIST;(absolute precondition)
  --POSTCONDITION : if not PRECONDITION
  --                  then raise NON_LISTED_ADDRESS;
  --                end if;
  --                if N>1
  --                  then N := N-1;
  --                  else the <AD,1> tuple is removed from the LIST;
  --                end if;
  function IN_LIST (AD : ADDRESS;LIST : ADDRESS_LIST) return BOOLEAN;
  --returns (<AD,N : POSITIVE> is in LIST);
  --and FALSE otherwise.
private
  type ADDRESS_LIST is new INTEGER;--temporary
end ADDRESSES;
```

Figure 4-22. ADDRESSES Specification

4.5.5.3 An Example of Abstract Data Type Which Involves Several Transformations

a. Specification

The specification of abstract data type BROADCASTING is in figures 4-23 and 4-24. BROADCASTING is built on ADDRESSES. ADDRESSES is purely sequential (an object of the type will never be used by several concurrent tasks); its specification, therefore, includes no Pr-T net and we will pay no interest to its Ada implementation for which the techniques presented in this paper are useless.

BROADCASTING allows the declaration of objects identified by the type MESSAGE_CARRIER which can be used to broadcast a single message. This abstract data type exports three operations involving a MESSAGE_CARRIER designated by the parameter CARRIER:

1. ROADCAST: broadcasts a message (MESS) to a set of receivers sharing the address (AD). BROADCAST is modelized by transition TB in figure 4-24.
2. RESET: invalidates the carried message. RESET is modelized by transition TRS on figure 4–24.
3. RECEIVE: used to receive a message (parameter MESS) giving the caller's address (AD). The caller waits until the CARRIER carries a valid message with the indicated address (AD).

```
with ADDRESSES;
generic
   type MESSAGE is private;
   type ADDRESS is (< >);
package BROADCASTING is
   package LOCAL_ADDRESSES is new ADDRESSES (ADDRESS);
   use LOCAL_ADDRESSES;
   type MESSAGE_CARRIER_STATE is
    record
      MESS : MESSAGE;
      MESSAGE_PRESENT : BOOLEAN := FALSE;
      AD : ADDRESS;
      LIST : ADDRESS_LIST;
   end record;
   type MESSAGE_CARRIER is limited private;
   --A MESSAGE_CARRIER (MC) encapsulates a MESSAGE_CARRIER_STATE (MC.MCS);
   procedure BROADCAST (MESS :MESSAGE;AD : ADDRESS;
                                    CARRIER : in out MESSAGE_CARRIER);
   --POSTCONDITION :
   --    (CARRIER.MCS.MESS = MESS) and
   --    (CARRIER.MCS.MESSAGE_PRESENT) and
   --    (CARRIER.MCS.AD = AD);
   procedure RESET (CARRIER : in out MESSAGE_CARRIER);
   --POSTCONDITION : not CARRIER.MCS.MESSAGE_PRESENT;
   procedure RECEIVE (MESS : out MESSAGE;
                                CARRIER : in out MESSAGE_CARRIER;
                                AD : ADDRESS);
   --PRECONDITION:
   --    (CARRIER.MCS.MESSAGE_PRESENT) and
   --    (CARRIER.MCS.AD = AD);(waiting precondition)
   --POSTCONDITION : MESS = CARRIER.MCS.MESS;
private
   type MESSAGE_CARRIER is new INTEGER;--temporary
end BROADCASTING;
```

Figure 4-23. BROADCASTING Specification (Textual Part)

When a message is received by one or several RECEIVE callers, it remains valid on CARRIER.

RECEIVE is modelized by the transition sequence

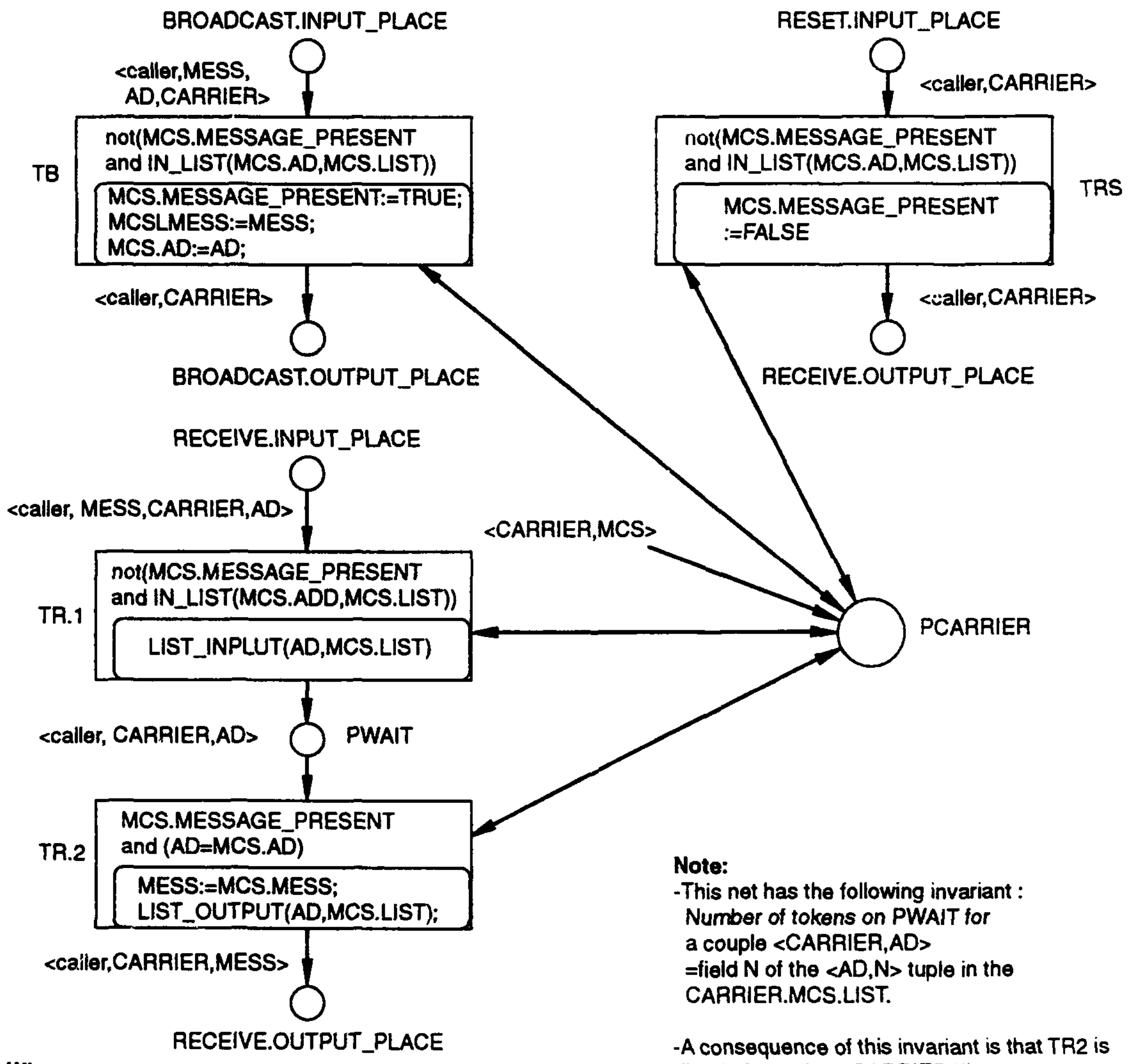
TR.1 (making the caller wait for a message),

TR.2 (making the caller receive the message) in figure 4-24.

Each MESSAGE_CARRIER object is modelized by a token on place PCARRIER in figure 4–24. Such a token has two fields:

1. CARRIER: designator of the MESSAGE_CARRIER object.
2. MCS: logical structure of the object, of type MESSAGE_CARRIER_STATE having four fields.
   (a) MESS: the carried message.
   (b) MESSAGE_PRESENT: a boolean denoting the validity of the message.
   (c) AD: the address of the addressees.
   (d) LIST: a list of tuples <address, number (>0) of potential receivers waiting at this address>.

The abstract data type ADDRESSES is used to build this field LIST. The note at the bottom of figure 4-24 gives an invariant of the Pr-T net which supports the proof of the interesting properties of the abstract data type BROADCASTING. This will not be developed here.


BROADCAST.INPUT_PLACE
<caller,MESS, AD,CARRIER>
TB
not(MCS.MESSAGE_PRESENT and IN_LIST(MCS.AD,MCS.LIST))
MCS.MESSAGE_PRESENT:=TRUE; MCSLMESS:=MESS; MCS.AD:=AD;
<caller,CARRIER>
BROADCAST.OUTPUT_PLACE
RESET.INPUT_PLACE
<caller,CARRIER>
not(MCS.MESSAGE_PRESENT and IN_LIST(MCS.AD,MCS.LIST))
TRS
MCS.MESSAGE_PRESENT :=FALSE
<caller,CARRIER>
RECEIVE.OUTPUT_PLACE
RECEIVE.INPUT_PLACE
<caller, MESS,CARRIER,AD>
<CARRIER,MCS>
TR.1
not(MCS.MESSAGE_PRESENT and IN_LIST(MCS.ADD,MCS.LIST))
LIST_INPLUT(AD,MCS.LIST)
PCARRIER
<caller, CARRIER,AD>
PWAIT
TR.2
MCS.MESSAGE_PRESENT and (AD=MCS.AD)
MESS:=MCS.MESS; LIST_OUTPUT(AD,MCS.LIST);
<caller,CARRIER,MESS>
RECEIVE.OUTPUT_PLACE

Where:
MESS; MESSAGE;
AD: ADDRESS;
CARRIER: MESSAGE_CARRIER;
MCS: MESSAGE_CARRIER_STATE;

Note:
-This net has the following invariant : *Number of tokens on PWAIT for* a couple <CARRIER,AD> =field N of the <AD,N> tuple in the CARRIER.MCS.LIST.

-A consequence of this invariant is that TR2 is firable for a given CARRIER iff :

CARRIER.MCS.MESSAGE_PRESENT
and IN_LIST(CARRIER.MCS.AD,
CARRIER.MCS.LIST) .

Figure 24. Broadcasting Specification (Pr-T Net Part)

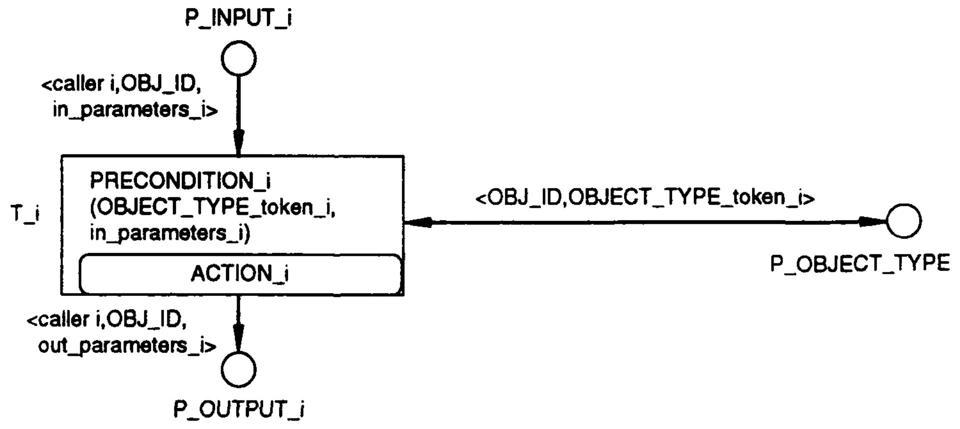
b. Implementation in Ada

This example of abstract data type BROADCASTING was chosen because it presents two main points of interest:

1. The specification is complex enough (i.e., far enough from the Ada semantics) to make its implementation by classical techniques almost untreatable; but, on the contrary, we are going to see that this implementation is made relatively simple through our transformational approach.
2. Transitions TB, TRS, TR.I in figure 4-24 map the first specification scheme and can be developed using our basic transformation; this is not the case with TR.2, since TR.2 precondition (MCS.MESSAGE_PRESENT and (AD = MCS.AD)) does not depend only on the state of the involved object (MCS), but also on an input parameter (AD).

The transformations which are necessary to bring this module from the abstract data type specification to an intermediate statr where the basic transformation is fully applicable remain rather simple while being representative of our transformation set as a whole.

The transformation to be applied to TR.2 is defined by figure 4-25 (input scheme) and figure 4-26 (output scheme). The basic principle of this transformation is to remove the precondition and to transform transition T_i, which uses an object with a precondition depending on an input parameter (fig. 4-25), into a macrotransition (T_i2, T_i3, T_i4, T_i5 on fig. 4-26) using this object with no precondition. This macrotransition computes the expression which was in the precondition (T_i2) and, according to the result (OK_ACTION_i) being true or false, it performs the action (T_i3) or it creates a waiting task WAITER_i and adds it to a set of waiting tasks (T_i4).
When this macrotransition, which ends with T_i5, has been fired, the operation corresponding to transition T_i terminates if the corresponding action has been performed (T_i9); or the caller waits for the waiting task transmitting the end_of_wait signal (T_i.6, T_i7, T_i.8) issued to the waiting task when the precondition is set true. (We will see a bit later how this signal is issued through places P_W.1, P_W.2. This leads to a new attempt to perform the action by looping back on T_i.2. The detail of the data structures supporting the waiting tasks and the sets of waiting tasks is beyond the scope of the present paper.

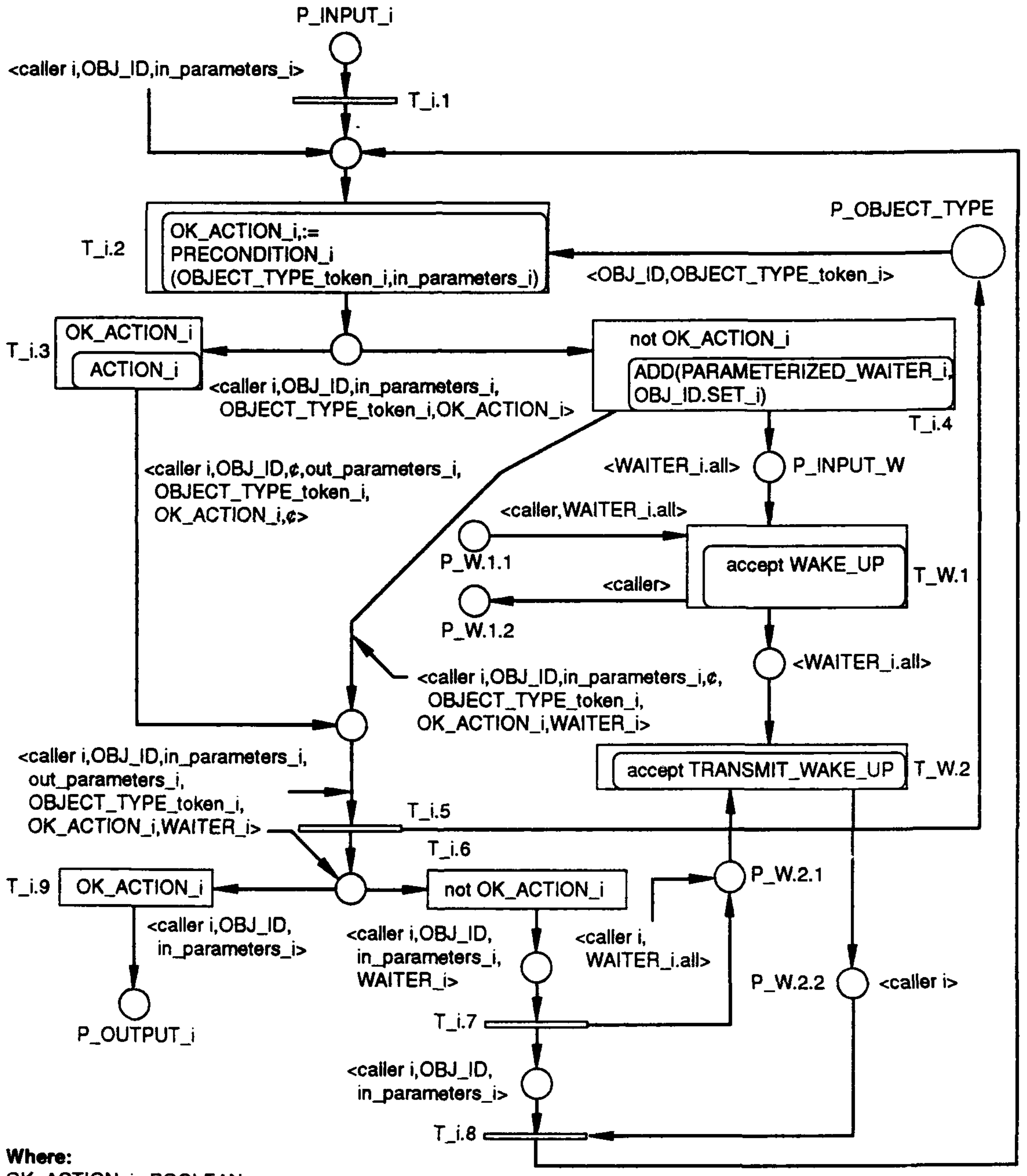

P_INPUT_i
<caller i,OBJ_ID, in_parameters_i>
T_i
PRECONDITION_i (OBJECT_TYPE_token_i, in_parameters_i)
ACTION_i
<OBJ_ID,OBJECT_TYPE_token_i>
P_OBJECT_TYPE
<caller i,OBJ_ID, out_parameters_i>
P_OUTPUT_i

```
package OBJECTS is
  type OBJECT_TYPE IS ...;
  OBJECT_INITIAL_STATE : constant OBJECT_TYPE := ...;
  type OBJECT_INDENTIFIER is limited private;
  --
  --
  procedure OPERATION_i (OBJ_ID : in out OBJECT_IDENTIFIER;
                         [formal_in_parameters_i]
                         [formal_out_parameters_i];
  --PRECONDITION : {PRECONDITION_i};
  -- waiting precondition depending on the in_parameters;
  --POSTCONDITION : {POSTCONDITION_i};
  --
  --
private
  --not a part of the specification
end OBJECTS;
```

Figure 25. Input Scheme of the Transformation Applicable to TR.2

We now see that the macrotransition (T_i.2, T_i.3, T_i.4, T_i.5) can be mapped on the first specification scheme since its precondition, being identically true, does not depend on any input parameter. The instantiation of the scheme in figure 4-26 for the T_i ⇒TR.2 mapping can now be implemented in Ada since:

1. The macrotransition (T_i.2, T_i.3, T_i.4, T_i.5) can be transformed into an Ada program fragment through the application of the basic transformation.



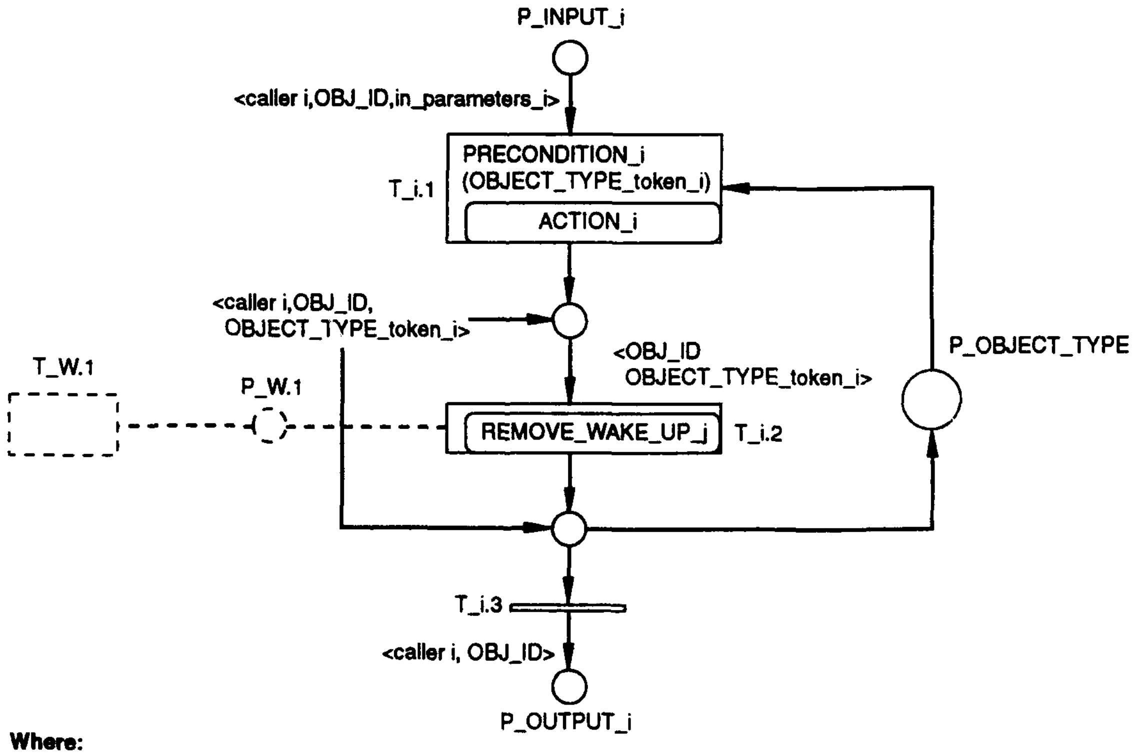
Where:
OK_ACTION_i : BOOLEAN;
WAITER_i : a waiter task to be detailed;
PARAMETERIZED_WAITER_i : an object grouping a waiter task and the input parameters of the operation which implies the wait;
OBJ_ID.SET_i : a set of parameterized waiters(1 set for each (OBJ_ID,OPERATION_i));
ADD(PARAMETERIZED_WAITER_i,OBJ_ID.SET_i);
PRECONDITION : PARAMETERIZED_WAITER_i is not in OBJ_ID.SET_i;
POSTCONDITION : PARAMETERIZED_WAITER_i is in OBJ_ID.SET_i;

Figure 26. Output Scheme of the Transformation Applicable to TR.2

2. The waiting tasks (transition sequence T_W.1, T_W.2) can be implemented as tasks belonging to an Ada task type having two entires and performing a sequence of two successive "accept" instructions which respectively refer to each of the two entries.
Because of the transformation which is applied to TR.2, each time a task calling the RECEIVE operation computes the precondition of TR.2 to false, a waiting task is created, which immediately engages itself in a wait, and this waiting task is added to a set of waiting tasks which is specific to the considered object and the considered elementary operation (TR.2). Each other operation, which may set true the precondition of TR.2, must remove the waiting tasks (which the precondition of TR.2 holds) from the corresponding set of waiting tasks and signal the tasks to end their wait. The only operation which may set true the precondition of TR.2 is BROADCAST (transition TB).

A transformation must then be applied to TB which has the above mentioned aim. The input scheme of such a transformation is the same as the input scheme of the basic transformation (to which TB could be mapped) and its output scheme is in figure 4-27. The macrotransition (T_i.1, T_i.2, T_i.3) in figure 4-27 is implementable in Ada by applying the basic transformation abstract data type BROADCASTING is implemented in Ada by using the program transformations demonstrated here provided that:

1. The data structures for the output schemes of these transformations are implementable in Ada.
2. The transformations applied to TR.2 and TB are correct as a whole.



Where:
REMOVE_WAKE_UP_j (j to be mapped to the index of TR2) :
Removes from the set OBJ_ID.SET_j,conceming the OPERATION_j on the object OBJ_ID, all the parameteriazed waiters W the parameter (W.I_P) of which satisfies PRECONDITION_j(OBJECT_TYPE_token_i,W.I_P) and wakes up these waiters.

Figure 27. Output Scheme of the Transformation Applicable to TB

On point a, the implementation of the data structures in Ada is easy and classical. For this implementation, standard set handling software components can be used which can be found, for example, in (ref. 4-45). We see here an interesting example of a possible mix between the transformational approach and the component reuse approach to software generation.

On point b, it is rather easy to give a proof scheme for the transformations applied to TR.2 and TB. at least if we accept a nonmechanizable proof scheme, but this development is beyond the scope of the present paper.

4.5.6 Current Status of Work

4.5.6.1 Comparison With Other Related Works

In the past few years, there have been distinct efforts for implementing concurrent systems specified by PETRI nets into programs in a high-level language (Ada most often) supporting asynchronous concurrent processes.

A first set of works (refs. 4-46 and 4-47) are closely related one to the other. They are rather different from the work presented here as follows:

a. Their PETRI net specifications are not already structured as asynchronous processes using common objects. So, the main part of their work is devoted to finding PETRI net transformations structuring the PETRI nets as asynchronous processes using common objects.

b. They have mainly considered simple place-transition nets at the specification level and are only starting to consider higher level nets. They did not consider (as we did here) the case of synchronizing transitionshaving any form of precondition that bears on, and possibly relates to, their input tokens.

c. Their PETRI nets have restrictive safety enforcing properties preventing the creation of dynamic processes and objects.

Apparently, the differences between the reference work and the work presented in this report can be explained because the aim of the reference works is mainly the implementation of communication protocols rather than the more general software system specifications as in this paper.

Another work (ref. 4-48) considers system specifications given in PROT nets. PROT nets are high-level PETRI nets, very close to colored PETRI nets or predicate-transition nets, but with restrictive safety and liveness enforcing properties (they are structured as cyclic static processes). Their implementation in Ada is very different from the present implementation in that their implementation is process-oriented (ours is object -oriented).

In addition to having each process on the net implemented by an Ada task, the reference work also considers each "synchronizing transition" (i.e., each transition having more than one input or output place) as being a process to be implemented by an Ada task.

4.5.6.2 Work Remaining at the End of 1989

a. The set of specification schemes for elementary operations presented in section 4.2.2 seems general enough to specify any abstract data type exporting synchronization/communication properties.

However, it would be nice to gather a set of different abstract data type specifications using this set of specification schemes to demonstrate that quality.

b. The set of transformations presented in section 4.3 is complete in that any abstract data type specified using our elementary operation specification scheme can be implemented in Ada through transformations in this set.

Requirements need to be issued for tools to support these transformations. However, realization of such tools will require more manpower than has been available during this development and an opportunity to realize these tools may not be practical in the near future

4.6 APPLICABILITY

In this section we examine the available industrial applications of the techniques so called transformations. We first of all introduce the way some large companies have decided to teach and to measure the availability of such techniques. We then present two applications, one of them being straightforwardly involved in Guidance and Control.

All the material presented so far pertains to experimental uses of program transformations in the industry.

No complete transformation system is already available for an operational use in the industry, but some formal specification (the starting point for program transformations) tools are already available and we next give two examples of such tools. Finally, we present the influence of the transformational approach on the Software Life Cycle.

4.6.1 Industrial Applications of Program Transformations

4.6.1.1 Teaching the Use of Transformations

A lot of software houses and large companies are looking carefully to these techniques. They participate to a lot of seminars, courses, conferences, and so on. On January 88, a one day conference about "formal specifications and their implications in the industrial world" has shown the interest of the industrial companies [afc88].

More important is the paper [Wor86] in which a nice experiment is reported. Indeed IBM itself in its laboratory at Hursley (UK) has experimented the teaching of formal methods of specification to development programmers. The language chosen was Z, a specification language widely used in Great-Britain.

The experiment itself has been conducted with people of varying degrees of experiences and background. The expected result of the course was to see how a formal specification can be used in an industrial environment, in redeveloping an already available module in CICS. Their main conclusions were:

a. Formal methods use mathematics that are relatively simple.

b. Formal methods can be enthusiastically received.

c. The technology being introduced will benefit from exposure to an industrial environment.

d. Education provided by consultants might need adaptations to meet industrial needs.

e. Education in reading documents with formal text must be given to a wide audience.

f. Users of an emerging technology need to see how it can be integrated into a complete software development process.

The last point (f) involves the transformations which are used in order to transform the initial specification into real programs.

4.6.1.2 Comparing Transformations

The second industrial examples concerns the BULL company. They reported in [BM88] part of the experiment they have conducted in their lab.

As for IBM, the main problem was to measure the impact of formal techniques on the software development process in rewriting part of GCOS-7, an operating system. The main difference relies on the fact that the people involved in such a development have been taught several formalisms: Z, VDM, B which belong to the world of formal specifications, and EIFFEL which is an Object-Oriented language presenting also some interesting formal specification aspects. Thus the two kinds of languages are not exactly at the same level. Nevertheless the conclusions they have drawn for the specification languages are of primary importance. They are comparable to those ones got in the IBM lab. Now that it has been proven that formal specifications are available, what about real developments?

4.6.1.3 Developing Software With Transformational Systems

We can report on two large examples of industrial software development with a transformational system. Because nothing has been officially published, no references are given.

Both of them are based on B [Abr88] a tool developed by J. R. Abrial. Very shortly described, B offers the users the capability to develop and use any kind of formal language. It can be seen as a support environment for the definition of a specification language and for its uses in a transformational process. Of course B integrates a lot of nice features for transforming the initial specification in an assisted manner and consequently to obtain a proved and correct program at the end.

The first example concerns the development of census of the French population in 1990. The development itself started a few months ago with a team of 10 people who have been taught B. After a while (less than 5 months) all the application was developed and formally verified. The first example shows the availability of the tool.

The second one is more concerned with our subject: Guidance and Control. The French company RATP (managing the Underground in Paris) decided a few years ago to automatize the traffic. Thus they have decided to ask a software house to realize an embedded system on each train able to drive automatically the train. The main trouble is related not to the feasibility of such a system but to its correctness. What can be said about it? The only right solution was to use a formal specification language and formal developments. In other words, because all the other methods failed, they have been obliged to demonstrate (in the mathematical sense) that every step of the development is correct versus the correctness of the initial specification.

4.6.2 Specification Systems Ready for an Industrial Use

At least two languages allowing formal specifications and available to the industrial world do exist: AnnA and EIFFEL.

AnnA [LvH85] stands for Annotated Ada. AnnA is an extension of the Ada language to provide facilities for formally specifying the intended behavior of Ada programs. It was designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation could be applied to Ada programs. For instance:

```
function IS_EMPTY(S:in SET) return BOOLEAN;
--| where
--|   return CARDINALITY(S)=0,
--|   out(for all X:ITEM ⇒ not IS/_MEMBER(X,S));
```

The first statement represents the name of an Ada function which checks that a set is empty. The first AnnA annotation specifies that the result of the function will be the BOOLEAN such that CARDINALITY of S is equal (or not) to 0. The second annotation specifies one way to define an empty set.

The EIFFEL language provides similar annotations. For instance, the *sqrt* function can be defined as follows:

```
sqrt (x, epsilon:REAL) IS
    REQUIRE
        X >= 0;
        epsilon >= 10^(-6)
    DO
        ... ...
    ENSURE
        abs (Result ^2-x) <= epsilon^2
    END sqrt;
```

In EIFFEL, the statement REQUIRE (respectively ENSURE) introduces the specification of a PRE-CONDITION (respectively POST-CONDITION) to be satisfied by the parameters of the function *sqrt*. When any of the assertions are violated an exception is raised.

What is important with the two languages is that they are already available for industrial applications. EIFFEL is marketed by the Interactive Software Engineering company, AnnA is under development. A first full AnnA compiler is available at the Stanford University.

4.6.3 Influence on the Software Life Cycle

The classical "waterfall" model of the Software Life Cycle is not applicable to software development using formal specifications and program transformations. The successive software design and costing phases no longer exist. It seems better to use the model presented by R. Balzer in [Bal85] in which from informal specification we can derive formal ones. They can serve as a first prototype for validating the specification. Then by applying transformations we can derive after several steps a source program.

4.7 LIMITATIONS

Although a few industrial projects have already used them, formal specifications and program transformations are not yet widely used due to some limitations which are developed in this section.

4.7.1 Inadequacy of the Present Software Development Context

Industrial companies have invested a lot of resources in implementing the present software development model. They have built or purchased a number of software tools and they have recruited large software development teams.

Software tools supporting formal specifications and program transformations are very different and more complex than the present tools supporting requirement analysis and software design.

Most people in the present software development teams are "analysts" or "programmers" who have been trained to use the present software development techniques and tools and very few people in these teams have a basic theoretical computer science and software engineering background.

Handling formal specifications and program transformations, on the contrary, requires smaller teams composed of high level software engineers and no "analysts" or "programmers."

4.7.2 Lack of Industrial Tools and Environments

The main drawback of the formal approach is the present lack of support from industrial environments.

Even in the case of Ada programming support environment (APSE) on which large amount of funds and efforts have been spent in order to meet the needs of the defense and aerospace industries, no tool supporting formal specifications is already available.

Among the other industrial environments, only the EIFFEL environment offers a (very limited) support for formal specifications.

However, as we have seen it previously, a number of research projects have undertaken the development of prototype tools to support formal specifications and program transformations.

This lets us hope that industrial tools in this area may be really available before this century is over.

4.7.3 Correct Codes Obtained From Formal Specifications and Program Transformations Does Not Ensure, by Itself, Correct Program Execution

As we have previously seen, program transformations allow us to generate code in a high level language (such as Ada) which is "correct by construction" in the sense that it automatically complies with the properties which were present in the formal specifications from which the code was generated.

This correct high level language code has still two major transformation steps to go through before it can be executed on some machines:

a. Transformation from high level language code to "logical machine" code;
b. Implementation of the "logical machine" code on a physical hardware machine.

Clearly, these two steps are beyond what is generally meant by "code generation" and consequently they are also beyond the scope of the present report. It may, however, be interesting to briefly report here that formal techniques also apply in these areas and that this application of formal techniques is the subject of important research programs.

The starting point for step a is a formal definition of the high level language. Considering only the example of Ada, several works have been performed to achieve a partial or complete formal definition of the language.

The most notable are:

a. An almost complete draft formal definition of Ada which has been worked out under the European Commission MAP program in 1985-87. This work was a technical success and it has been presented in a number of occasions, including a specific one-week tutorial presented in RENDE (CALABRIA_ITALY) in October 1987. However, this formal definition of Ada is very large and a significant amount of resources, beyond what was spent on the MAP program, would still be needed to publish it in a usable form.

b. Definition and formalization of subsets of Ada such as:

    1. "SPARK" [CARDE 87/4-58] or "Safe Ada" [HOLW187/4-57] which are "safe", easily formalizable subsets of ADA,
    2. More or less complete formal definitions of Ada tasking; see, for example [dB 83/4-44] which is also used in the material presented in section 4.5.5.
    3. Steps a and b also require a formalization of the logical machine and step b furthermore requires a formalization of the mapping of the logical machine on a physical hardware machine.

       Works have also been undertaken in these areas; see, for example, the British Ministry of Defence project VIPER [? 4-58], but there is still a long way to go before machines of widely usable size and power can be formalized.

4.7.4 Limitations of the Asynchronous Process Paradigm for Specifying Concurrent Systems

In section 4.5 we showed that it is relatively easy to give an abstract asynchronous process model for a concurrent system which naturally expresses the functional properties of the system and to implement such an abstract model into some efficiently compilable/executable code (such as Ada) using program transformations.

A serious limitation of this way of developing concurrent systems is that the asynchronous process model captures the temporal properties much less well and naturally than the functional properties. Simple delays, simple time-outs can be easily expressed in the asynchronous process model, but absolute time constraints (such as: a given process terminates either when it completes its normal execution or when a given delay has elapsed or a given event has occurred) cannot be expressed since they are contradictory with the very definition of asynchronous processes (which "proceed independently except on points where they synchronize").

"Synchronous models" of concurrent systems have been developed since the mid-1980s to overcome this limitation.

The general principle of synchronous models is to decompose systems in concurrent processes or data flows which are synchronized by cycles, or signals. These systems rely heavily on a "strong synchrony hypothesis" stating that all synchronization and control operations take no time at all; they are "immediate".

ESTEREL [BE4ON88/4-59] is a good example of a synchronous model which has been developed and supported by tools to a point where it can be used by the industry, at least for experimental purposes. In ESTEREL, one can define concurrent processes which one synchronized by "signals" coming from the external world or emitted by the processes. When a signal is emitted, all the synchronizing operations which were waiting for S are performed at once, in the "instant" of emission of S (following the "strong synchrony" hypothesis). An absolute time constraint can be expressed the following way in ESTEREL:

```
do MY_OPERATION upto MY_SIGNAL;        (1)
or:
do MY_OPERATION uptonext MY_SIGNAL;  (2)
```

In statement (1) MY_OPERATION is started only if MY_SIGNAL is not present at start-time and statement (1) lasts exactly until MY_SIGNAL is emitted (if MY_OPERATION completes before that, (1) waits until MY_SIGNAL is emitted; if MY_SIGNAL is emitted while MY_OPERATION is processing, then MY_OPERATION is abruptly terminated).

Statement (2) is similar to statement (1) except that MY_OPERATION is always started and statement (2) lasts until the next emission of MY_SIGNAL (not counting an emission occurring at statement (2) start-time). The operative part of an ESTEREL program may be written in Ada or C, properties may be proved on ESTEREL programs and there is a program transformation system implementing an ESTEREL program as a finite state machine which is generated either in Ada or in C.

Of course, synchronous models are still in their infancy and their use is for less well understood than the use of asynchronous process models, even in the case of ESTEREL which has accumulated the longest use experience.

4.8 REFERENCES

4-1 B. Boehm. Software engineering. *IEEE TSE*, C-25(12), 1976.

4-2 H. Partsch and R. Steinbruggen. Program transformation systems. *ACM Computing Surveys*, 15(3), 1983.

4-3 M. S. Feather. A system for assisting program transformation. *ACM-TOPLAS*, 4(1), 1982.

4-4 D. Wile, R. Balzer, and N. Goldman. Automated derivation of program control structure from natural language program descriptions. *SIGPLAN Not.*, 12(8), 1977.

4-5 D. Wile, R. Balzer, and N. Goldman. On the transformational approach to programming. In *Proceedings of the 2nd ICSE*, San Francisco, California, 1976.

4-6 D. S. Wile. Popart: Producer of parsers and related tools, rr-82-21. Technical report, ISI, Marina Del Rey, California, 1982.

4-7 D. S. Wile. Program developments: formal explanations of implementations. *CACM*, 26(11), 1983.

4-8 C. Green. A summary of the PSI program synthesis system. In *Proceedings of the 5th IJCAI*, Cambridge, Mass., 1977.

4-9 D. R. Barstow. On convergence toward a data base of programming rules. *ACM TOPLAS*, 7(1), 1985.

4-10 E. Kant. A knowledge-based approach to using efficiency estimation in program synthesis. In *Proceedings of 6th IJCAI*, Stanford, California, 1979.

4-11 J. Darlington. A synthesis of several sorting algorithms. *ACM TOPLAS*, 7(1), 1985.

4-12 Z. Manna and R. Waldinger. Dedalus. In *Proceedings of the NCC*, Anaheim, California, 1978.

4-13 J. Arsac. *La Construction de Programmes Structurés*. DUNOD, Paris, 1977.

4-14 J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM, Trans. Prog. Lang. Syst.*, 4(2), 1982.

4-15 H. Partsch. The CIP transformation system. In F. L. Bauer and L. H. Remus, editors, *Program Transformation and Programming Environments*, pages 305-322. Springer-Verlag, 1984.

4-16 F. L. Bauer and al. *The Munich project CIP, The wide-spectrum language CIP-L*, volume 1. Springer-Verlag, LNCS-183, 1985.

4-17 F. L. Bauer and al. *The Munich project CIP, The transformation system CIP-S*, volume 2. Springer-Verlag, LNCS-292, 1987.

4-18 V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy. A structured oriented program editor. A first step towards computer assisted programming. In *Proceedings of the Int. Computing Symposium*, Antibes, France, 1975.

4-19 J. Foisseau, R. Jacquart, M. Lemaitre, M. Lemoine, and G. Zanon. Program development with or without coding. In *Proceedings of the IFIP 80*, Melbourne, Australia, 1980.

4-20 J. Foisseau, R. Jacquart, M. Lemaitre, M. Lemoine, and G. Zanon. Le système SPRAC: expression et gestion des spécifications, d'algorithmes et de représentations. *Techniques et Sciences Informatiques*, 4(3), 1985.

4-21 R. Jacquart, M. Lemoine, and G. Zanon. Transformational tools used in a software environment. In P. J. Brown, editor, *Proceedings of the Software Engineering Conference 86*, pages 175-183. IEE-CS 6, London, 1986.

4-22 C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall International, 1980.

4-23 C. B. Jones. *Systematic development using VDM*. Prentice-Hall International, 1986.

4-24 P. Facon. Langages ensemblistes et transformations de programmes. In *Proc. of the 10th STFI-TUNIS*, May 1989.

4-25 B. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient ram code - a case study. *Journal of Symbolic Computation*, 4, 1987.

4-26 J. T. Schwarts, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets, an Introduction to SETL*. Text and Monographies in Computer Science-SPRINGER-VERLAG, 1986.

4-27 L.M. Abraido-Fondino. An overview of REFINE 2.0. In *The 2nd International Symposium on Knowledge-Engineering/Software Engineering*, April 1987.

4-28 H. Horgen, TOOLUSE: an advanced support environment for method-driven development and evolution of packaged software. In CEC-DGXIII, editor, *ESPRIT'85: Status Report*, pages 545-562. North-Holland, 1985.

4-29 J. Cazin, R. Jacquart, M. Lemoine, P. Michel, and P. Maurice. Method driven programming. In G. X. Ritter, editor, *IFIP 11th World Computer Congress*, August 1989.

4-30 T. T. Nguyen. On Jackson's structured programming method. Technical report, UCL, ToolUse.TD.TN88/1a, Feb. 1988.

4-31 M. Jackson. Structure-oriented programming. In P. Pepper, editor, *Program Transformation and Programming Environment*. NATO ASI Series, Vol. F8, 1984.

4-32 F. Vivares. Conception et réalisation d'un environnement d'aide à la conception de systèmes informatiques. Technical report, CERT-DERI, 1988.

4-33 G. Berthelot. Transformations and decompositions of nets. In *Advances in PETRI Nets 1986 Part 1*. LNCS 254, Springer-Verlag, 1987.

4-34 W. Reisig. Petri nets in software engineering. In *Advances in PETRI Nets Part 2*. LNCS 255, Springer-Verlag, 1987.

4-35 P. de Bondeli. Spécification de types abstraits pour programmes parallèles au moeyn de réseaux de Petri à prédicats, implantation en Ada. In *Journ' es AFCET-GROPLAN sur le Développement de Programmes pour Machines Parallèles*, 5 1989.

4-36 *PETRI Nets: Central Models and their Properties - LNCS 254*, 1987.

4-37 Reference manual for the Ada programming language, ANSI-mil-std/1815a-iso 8652. Technical report, 1987.

4-38 E. Perez. Simulating inheritance with Ada. *ACM Ada Letters*, 8(5), 1988.

4-39 C. M. Donaldson. Dynamic binding and inheritance in an object-oriented Ada design. In *Proc. of the Ada - EUROPE International Conference - MADRID*. The Ada Companion Series - CAMBRIDGE University Press, 1989.

4-40 J. P. Forestier, C. Fornarino, and P. Franchi-Zanettacci. Ada++, a class and inheritance extension for Ada. In *Proc. of the Ada - EUROPE International Conference - MADRID*. The Ada Companion Series - CAMBRIDGE University Press, 1989.

4-41 *PETRI Nets: Applications and Relationships to Other Models of Concurrency - LNCS 255*, 1987.

4-42 P. de Bondeli. On the use of semantic specifications for the verification and validation of real-time software. In *Proceedings of the 3rd IDA workshop on ADA verification*, 1986.

4-43 P. de Bondeli. Implantation en Ada de types de synchronisation et communication inter-taches specifies par des reseaux predicats-transitions. In BIGRE + GLOBULE, editor, *Proc. of the Conference Ada AFCET/ENST - PARIS*, 1987.

4-44 P. de Bondeli. Models for the control of concurrency in Ada based on predicate-transition nets. In Commission of the European Communities, editor, *Proc. of the ADATEC - ADA - EUROPE Conference - BRUSSELS, 1983.*

4-45 G. Booch. *Software Components with Ada*. BENJAMIN/CUMMINGS, 1987.

4-46 J. M. Colom, M. SILVA, and J. L. Villaroel. On software implementation of PETRI nets and colored PETRI nets using high-level concurrent languages. In *Proc. to the 7th European Workshop on Application and Theory of PETRI Nets - OXFORD (UK)*, June 1986.

4-47 B. Cousin, P. Estrailler, and M. Kordon. Generation of Ada code from a PETRI net model: an application. In *Proc. of the 3rd International Symposium on Computer and Information Sciences*, October 1988.

4-48 G. Bruno and G. Marchetto. Process translatable PETRI nets for the rapid prototyping of process control systems. *IEEE Trans. on Software Engineering*, 12(2), February 1986.

4-49 In *Impact des Spécifications Formelles sur le Monde Industriel*. AFCET, 1988.

4-50 J. Wordsworth. Teaching formal specification methods in an industrial environment. In D. Barnes and P. J. Brown, editors, *Software Engineering* 86, pages 43-51. Peter Peregrinus Ltd., 1986.

4-51 P. Behm and F. Meija. Un exercice de spécification avec VDM. In 4'eme Colloque de GENIE LOGICIEL, AFCET, October 1988.

4-52 J. R. Abrial. Une approche formelle des logiciels. Technical report, 1988.

4-53 D. C. Luckham and F. W. von Henke. An Overview of AnnA, a Specification Language for Ada. *IEEE Software*, 2:9-23, March 1985.

4-54 R. Balzer. A 15 year perspective on automatic programing. *IEEE Software*, 11:1257-1268, November 1985.

4-55 Krieg-Bruckner and Hoffman . PROgram development by SPECification and TRANsformation (PROSPECTRA) to be published in 2 volumes of LNCS Springer Verlag 1991.

4-56 B. Carre, T. Jennings. SPARK-The Space Ada Kemel. Department of Electronics and Computer Science, Universil F Southamptom UK.

4-57 R. Holzapfel, G. Winterstein - Ada in Safely Critical Applications. In Proc. of the Ada-Europe Intemation Conference MUNICH. Ther Ada Companion Series, Cambridge University Press, 1988.

4-58 Re Feng on VIPER to be provided by K. Helps.

4-59 G. Beery, G. Gonthier. The ESTEREL Synchronons Programming Language: Design, Semantics, Implementation INRIA Research Report No. 842-May 1988.

BIBLIOGRAPHY

E. Astesiano and G. Reggio. The SMOLCS Approach to the specification of concurrent systems. *Distributed Systems on Local Network*, 2, 1985.

J. P. G. Bames. *RGL/2 Design and Philosophy.* HEYDEN - Intemational Topics in Science, 1976.

M. S. Feather. A survey and classification of some program transformation approaches and techniques. In *Program specification and transformation*, BadTolz, FRG, 1984. IFIP TC2/WG2.1, Nort-Holland

C. A. R. Hoare. *Communicating Sequential Processes.* CACM, 21(8), 1978.

C. A. R. Hoare. *Communicating Sequential Processes.* PRENTICE HALL, 1985.

LTR3 - Manuel Officiel de Référence (indice 2), July 1985.

R. Milner. *A Calculus of Communicating Systems - LNCS 92.* SPRINGER VERLAG, 1980.

R. Milner. *Letters on a Calculus of Communicating Systems - LNCS 197.* SPRINGER VERLAG, 1984.

Programm'er Sprache PEARL (Full PEARL) - DIN 66253 Teil 2, November 1980.

CHAPTER 5

FOURTH GENERATION LANGUAGE APPROACH TO SOFTWARE GENERATION

5.1 OVERVIEW

5.1.1 Background

Distinctions are often blurred, but the following are characteristics of the first three language generations:

First generation languages were those which dealt exclusively, or almost exclusively, with arithmetic and store manipulative operations, control register operations and store locations.

Second generation languages introduced symbolic identifiers for variables which were no longer assigned by the programer to a specific location. The programmer was virtually freed from concern with store location reference numbers, working instead with a set of names and labels of his own choosing which could have some mnemonic correspondence to elements of the problem being addressed.

Third generation languages, roughly stretching from Fortran to Ada, was typified by for-do loops, if-then-else conditionals and the generic handling of arrays, tables and lists. In later years they have incorporated a variety of constructs including support for typing, interface concerns, parallelism and richer data structures.

The nearer one gets to current practice, the less clear is the perspective. The characteristic features of the third generation described above is rather more of a rag-bag list of properties than the coherent single concept which delineates a second generation language.

With the fourth generation, as yet in its infancy, the problem is intensified. MacNicol (ref. 5-1) defines a fourth generation language (4GL) as follows:

> "The term "Fourth Generation Language" is slightly misleading. It is a label that can be applied to a wide range of products, many of which are not what most people would recognize as "languages" at all. Traditional computer languages require the programmer to specify not only "what" is needed but also "how" the computer is to do it.
>
> The term 4GL was invented to label a growing number of products which enable a computer to be programmed by specifying only "what" is needed and leaving the "how" to the 4GL to sort out. The spread of products which can legitimately be labelled '4GL' includes: database enquire languages, application generators, spreadsheets, decision support software and analyst workbenches."

Whether this is an acceptable definition, and in particular whether the second paragraph is an adequate representation of the facts, will be discussed later in this chapter. What it does make clear is the general welter of confusion subsumed in the term "4GL".

Reed (ref. 5-2) goes so far as to say

> ".... a 4GL is any software product that automates the job of a programmer".

Schussel (ref. 5-3) gives a more useful classification

"The 4th generation can be principally thought of as the rise of the relational DBMS; programmer oriented 4GLs; information centered 4GLs; and decision support systems."

The three foregoing "definitions", which are characteristic of many more lead inescapably to the conclusion that the term 4GL, while unquestionably a useful tool in the marketing department of a commercial software house, has as yet no formal place in a control engineering department or in the Guidance and Control software domain.

5.1.2 Working Definition of the 4GL Approach

For use in consideration of automated Guidance and Control software generation, this paper makes the following definition:

The 4GL Approach to Automatic Generation of Guidance and Control Software

The 4GL approach to the automatic generation of software for guidance and control is that which has helpful and powerful interactive support from a workstation or networked workstations, which is so oriented to the users' capabilities and concepts of the application domain, that the time and effort needed to generate the software (and necessary related documentation and products) is dramatically reduced.

i.e. 4GL Approach - Fourth Generation Language + Software Engineering Support

This definition is illustrated in figure 5-1.

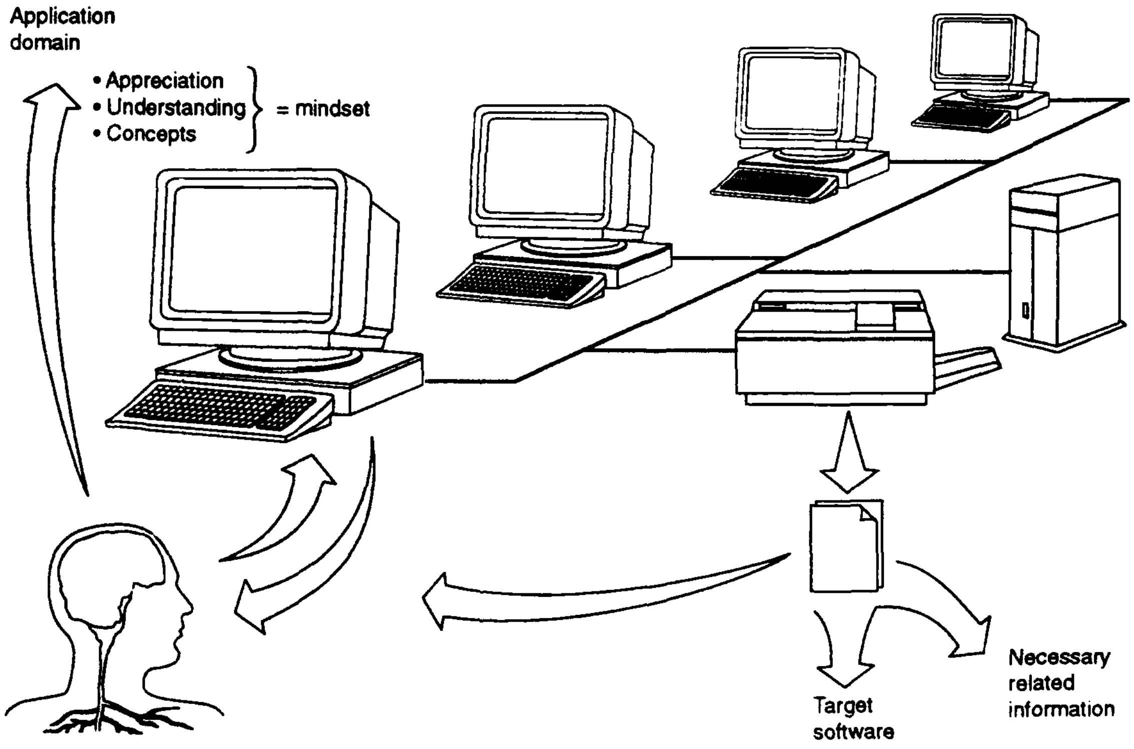


Figure 5-1. The 4GL Approach to Automatic Generation of Guidance and Control Software

The definition above embraces the key motivation for the 4GL approach which is to provide maximum automation with retention of a natural approach to the application.

5.1.3 Comparison With Objectives of Alternative Approaches in This Report

The four approaches to software generation described in Chapters 2 to 5 of this report have each a very different objective, and a different impact on the structure of the systems development teams.

Re-use of software aims to minimize the current waste of software effort whereby substantially similar design problems are tackled ab initio over and over again. It has relatively little impact on the control engineering department's workload or approach to the task, but should offer some improvement in overall product integrity since it increases the usage of software components which have already been proved in service. Software effort required and software timescale should be reduced.

By contrast, the expert system approach aims to reduce effort and timescales across the board. It captures domain specific knowledge. The software engineering of the project in hand is only one small part of the target of the expert system. Inasmuch as the expert system aims to augment or replace the software expert, while the 4GL approach aims to remove the mechanical aspects of the task, these two techniques are completely complementary.

Program transformation holds out the possibility of improvements in system integrity by supporting the equivalence of implementation and specification, either by direct forward automated or semi-automated transformations or by making possible formal verifications. It may also lead to gains in productivity by minimization of errors.

The 4GL approach primarily reduces the software engineering element of the program. If anything, the control engineering element might appear to be increased, since it now includes the software engineering, albeit in a much simplified form. However, this increase is balanced by the fact that some of the combination of system constructs, and all of the translation into software engineering terminology, is automated.

Successfully applied, the 4GL approach should offer:

a. An improved product since one communication barrier has been removed, and the system designer has a closer rapport with the final software.

b. Improved productivity, shorter timescales and a smaller total team.

c. A greater degree of consistency in the product.

d. Easier development and maintenance, which can be undertaken from a system engineering, rather than from a software engineering viewpoint.

5.1.4 Relationship of 4GL Approach to Current Software Generation Concepts

The 4GL approach can be considered in relation to two current software generation concepts - Integrated Program Support Systems (IPSEs) and Rapid Prototyping. Figure 5-2 illustrates these relationships.

5.1.4.1 Comparison With an IPSE Approach

The essence of an IPSE is the provision of an integrated set of generally applicable software engineering tools supporting the software generation process.

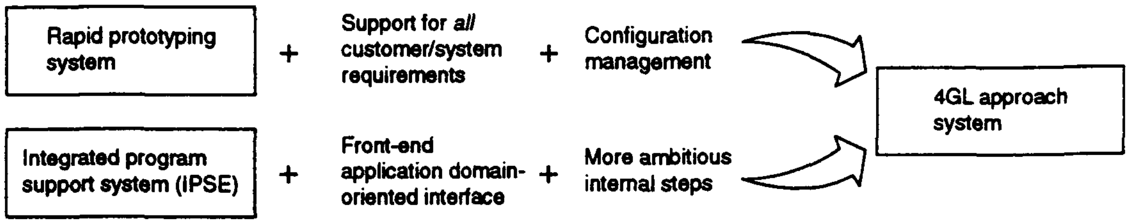

Rapid prototyping system
+
Support for *all* customer/system requirements
+
Configuration management
Integrated program support system (IPSE)
+
Front-end application domain-oriented interface
+
More ambitious internal steps
4GL approach system

Figure 5-2. Relationship of 4GL Approach to Current Software Generation Concepts (Migration Routes to a 4GL Approach System)

The 4GL approach would require a significant strengthening of the interface with the application domain. This would allow requirements to be selected (by simulating or otherwise evaluating alternatives) and securely capturing the requirements for further translation into an implementation.

The 4GL approach would also aim at having more ambitious internal steps in its automatic software generation process, requiring less attention on the part of the user to stages of translation and integration of the software being produced.

5.1.4.2 Comparison With Rapid Prototyping

The 4GL approach shares a number of features with the essentials of a Rapid Prototyping system, as typified by a Symbolics Genera Lisp environment, e.g. convenience of evaluating functional requirements, a user friendly operating system sometimes supported by application domain oriented packages.

Typically, however, Rapid Prototyping systems stop short of providing support for all customer and system requirements, particularly non-functional requirements. They are also not normally equipped to provide an adequate level of configuration management, which may in part be seen as a customer requirement, but is also a necessary feature of maintaining control in the development of software through evolutionary requirements, especially for protracted and large scale projects.

Rapid Prototyping is surely an inspiration to moving towards a fully 4GL approach (as defined in section 5.1.2).

5.1.5 Distinguishing Characteristics of the 4GL Approach

In summary the distinguishing features of the 4GL approach are:

a. Improved support for the interface between the application domain and the 4GL system user(s).
b. Concentration of integrated tools to give more power to the user(s) in undertaking the whole software generation task.
c. Minimizing the number of visible steps in the software generation process, i.e., the number of levels at which the user needs to exercise control.

5.1.5.1 Support for the User's Interface to the Application Domain

The support for the user's interface to the application involves:

a. Automated analysis of the problem area.
b. Capture of the functional and nonfunctional requirements.

The first of these is straightforward in concept, although the ease with which such analysis can be automated and the necessity for it to be done at all varies widely.

The second is inherently difficult to do completely. Completeness of requirements cannot be guaranteed even with requirements specification at a single functional level, for obvious reasons. Still less can it be guaranteed when dealing with the satisfying of non-functional requirements. For functional specifications, self-consistency checks may be automatable, but with non-functional requirements, this is certainly not the case in general. Some discussion of functional and non-functional requirements may illustrate the problem.

5.1.5.1.1 Functional and Non-Functional Requirements

Approaches to automatic software generation which emphasize the importance of supporting the capture of requirements from the application domain, as the 4GL approach does, inevitably raise the question of the separation of WHAT needs to be done from HOW it is to be done. A brief discussion is appropriate.

The importance of distinguishing in software between WHAT and HOW has long been appreciated, e.g. as expressed e.g. by Dijkstra (ref. 5-4):

> "There is .. an abstraction involved in naming an operation and using it on account of 'what it does' while completely disregarding 'how it works'."

Swann (ref. 5-5) has stated (and this is just one statement of a widely held principle):

> "Good design practice demands that wherever possible we separate our design concerns. In particular we "separate the WHAT from the HOW". So, for a software system, we divide the development process into two major parts. First, we develop a specification: the WHAT, then we design and code: the HOW.
>
> "The aim of the specification is that it captures everything we need to know about the customer domain in a form which is free from any considerations of how it will be implemented."

It is, however, necessary to give some more careful consideration to these words WHAT and HOW. There is no definitive distinguishing characteristic which can separate WHAT statements from HOW statements. Indeed a statement which is a WHAT from one point of view may be a HOW from another.

> "Let us invite the captain of the cricket club to tea" sounds very like a pure WHAT statement. The fact that it arose from the thought "HOW shall I enhance my chance to be selected to bat against Twittering Magna" is tactfully overlooked by all concerned. A few days later the WHAT statement "Bill is coming round at 4:30 on Sunday. Will you make one of your special cakes?" is a statement of HOW the captain is to be invited to tea, and of WHAT he is going to be offered. The recipe for the cake, which was used for an exactly similar purpose when she wanted to sing Yum-Yum with the village operatic society,

comes from the Daily Telegraph Cookery Book and is, to all intents and purposes, a piece of reusable software.

As a more mundane technical example consider the transition of WHAT and HOW in example of figure 5-3.

• Example 1	
• Highest level requirement	Align vehicle to planned trajectory
• High level requirement	Set control surfaces in relation to misalignment
• Medium level specification	Evaluate control law transfer functions; drive actuator
• Example 2	
• Highest level requirement	Make displays adequate for safety
• High level requirement	Make display x attention getting
• Medium level specification	Make display x red
• Lower level specification	Set display color control to value 7

Figure 5-3. No Fundamental Difference Between Requirements and Specifications (One Man's "What" Is Another Man's "How")

Every statement is mainly HOW from the point of view of the makers of decision upstream of its occurrence, and mainly WHAT from the point of view of the downstream decision makers. 'Red' is the HOW of 'attention getting' and the WHAT of ' display color control value 7', etc. There are frequently many HOWS for each WHAT.

Thus functional and non-functional definitions change with levels of refinement. In considering the 4GL approach to automatic software generation it cannot be assumed that a neat demarcation can be made between functional and non-functional inputs.

5.1.5.1.2 Categories of User Information Input

It is useful to consider categorizing the inputs which the user of a 4GL system would need to make or have made for him by the system.

In the guidance and control field we would submit that in the present level of technology the most useful categorization is into four classes of decision:

a. Project policy decisions.

b. Systems engineering decisions, e.g.:

   1. Hardware configuration.
   2. Software functional requirements.
   3. Algorithms.
   4. External interface constraints.
   5. Timing constraints.

c. Software engineering decisions, e.g.:
   1. Modularization.
   2. Inter module data protocols.
   3. Data management, etc.

d. *Coding decisions.*

The degree to which these types of decisions are currently supported or likely to be satisfied is discussed in paragraphs 5.2 and 5.3. In general terms one may expect a 4GL approach to:

a. Provide no contribution to the project policy decision (except that the existence of an applicable 4GL system would influence the decision).
b. Assist in the capture of systems engineering decisions.
c. Undertake many software engineering decisions by default.
d. Normally make coding decisions.

5.1.5.2 Concentration of Integrated Tools

The concentration of integrated tools is not unique to the 4GL approach and will not be discussed at length. The characteristics generally emphasized in this matter for the 4GL approach are:

a. Networked workstations (where the size of the software generation demands more than a single workstation, as will normally be the case).
b. Large high resolution (and possibly color) screens capable of supporting high resolution graphics with windows and a natural user-friendly approach to the control of, and access to, software items and documents. Ability to see separate documents simultaneously. Convenient control of workstation by keyboard and mouse. High performance workstation for responsiveness to the user. Ample memory for immediate access and file storage. Good monitoring of resource usage. Good default capabilities with optional user over-rides, e.g. in file serving.

5.1.5.3 Minimizing the Number of Visible Steps in Software Generation

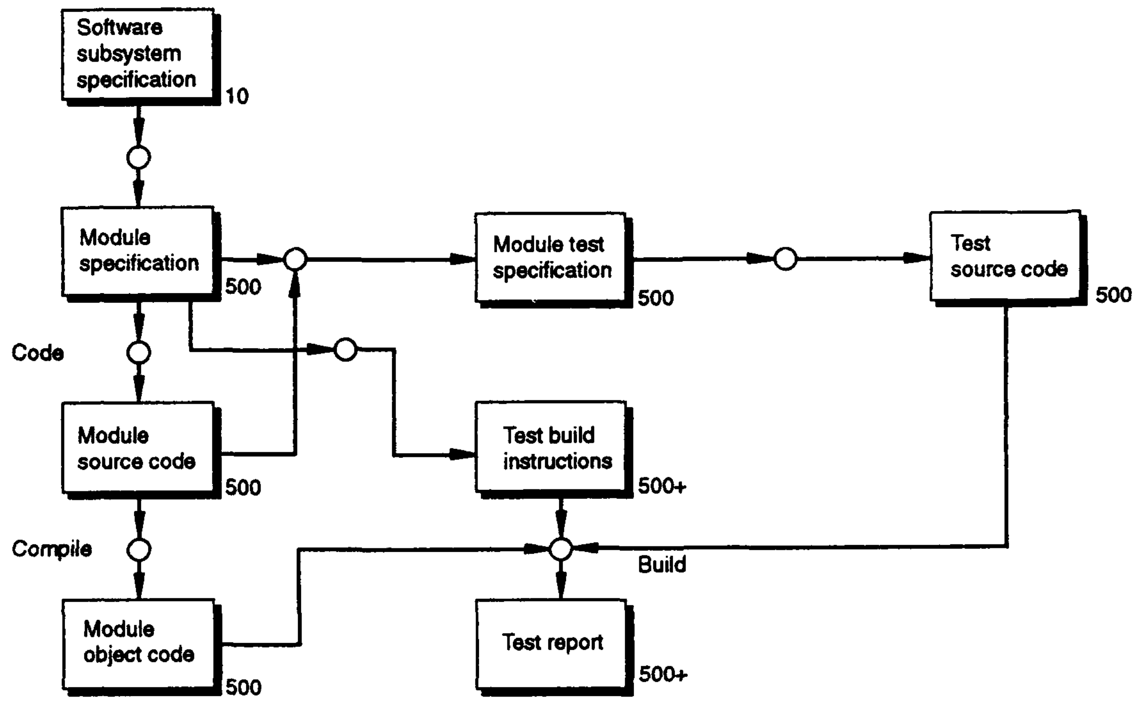
Some of the scope for productivity improvements is shown in figure 5-4. This shows that large fan-out from layer to layer of software in a 500 Kbyte target code guidance and control program, and indicates the large number of software items, most of which require detailed human attention. When problem reporting and consequent changes are considered, the effort required to develop the software is large.

The 4GL approach can be expected to give rise to significant improvements of software productivity. These are illustrated in figure 5-5.

5.1.6 Some Key Technologies Involved

The Shorter OED defines a language as "The whole body of words <u>and of methods of combining them</u> used by a nation, people or race." A computer language is generally expected to embrace not just definition of the "words" (the primitives) but also a definition of the syntax and meaning of combinations of words (the constructs).

Although a computer language can have a meaningful, and even a useful, existence before any automatic tools to handle it have been implemented, the usual minimum sine qua non for acceptance is a compiler. The



Assumes system with:
500 Kb target code
10b/SLOC (source line of code)
100SLOC/module
50 modules/subsystem

Note:
The numbers of items shown do not include versions and variants

Figure 5-4. Some of the Scope for Productivity Improvements

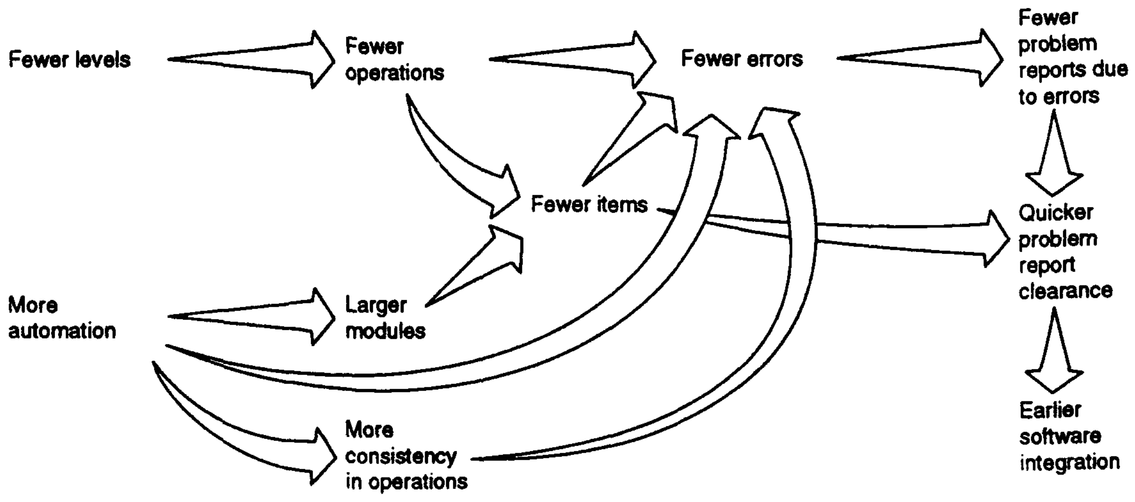

Fewer levels
Fewer operations
Fewer errors
Fewer problem reports due to errors
Fewer items
Quicker problem report clearance
More automation
Larger modules
Earlier software integration
More consistency in operations

Figure 5-5. Productivity Potential From Fewer Stages and Fewer Items

compiler takes a document (the word 'document' is used here to avoid use of the word 'text', which might imply the exclusion of documents in a graphical format) written in the language - the source code - and translates it into a lower level language - the object code.

In the ultimate ideal envisaged by this report, whereby control engineering requirements, for example, are translated into executable object code without further human intervention, there would be no need for the lower level languages, i.e. the software engineering language and the coding language, to make any concessions to human readability. Indeed, it is common practice in compiler design to introduce an intermediate language which is not necessitated by the functional requirement of the language and whose readability is not a primary design feature. This intermediate language is there solely to introduce a degree of modularity into the compiler: to separate the front end from the back end so that either source of object language can be changed with much less work than the writing of a whole new compiler.

However, experience in comparatively simple applications has shown that 100% automation is not always possible, and most compilers make some provision for inserting blocks of object code into the source code to solve some of the more intractable problems - a practice which, although sometimes difficult to avoid, generally undermines the feasibility and effectiveness of integrated toolsets.

The need to use some low level code inserts is likely to remain given the embedded nature of much guidance and control software and its degree of complexity. A consequence of this is to strengthen the need to permit human visibility (readability and comprehension) of the generated code in the output of compilers and for an automated system to support the understanding of such code in relation to the functional specification language designed for human understanding.

5.1.6.1 Primary Components of a 4GL System

With these premises, then, the requirement is for:

a. A system engineering language, comprising

   1. A set of application domain primitives, application domain constructs and a combinational syntax which make up the language proper.
   2. A set of tools for manipulating this language.
      Ideally this set of tools will be geared to the use of the systems engineer, with a minimum of need for expertise in subsequent stages of the program. They should support the incremental, interactive development of the system design by a team, rather than by an individual, and support configuration control, multi-viewpoint reporting, simulation, validation, and all of the usual desiderata of a computer aided design package. There is a strong body of opinion which suggest that these tools, and indeed the language itself should be graphical, or at least should be capable of being used from a largely graphical viewpoint.
   3. A compiler which translates the language of 1. above into the language of b.1 below.
      In a sense this compiler includes a large body of reusable code segments. Indeed it is possible to regard any compiler as containing a hierarchical structure of reusable, parametrically addressed code segments and structures. As the level of abstraction of the language rises, this view becomes more significant.

b. A software engineering language, comprising

   1. A set of software engineering primitives, software engineering constructs and a combinatorial syntax.
   2. A set of tools for manipulating this language.
      Where appropriate, these should be linked to the tools of a.2., e.g. in the configuration control and reporting aspects, but they should also support software development ab initio.

3. A compiler whose source is the language of b.1., and whose object is the language of c.

c. A high level computer language with all the usual expected attributes.

d. Run-time executives for target machines and target distributed network.

It is worthwhile here to refer to "the myth of the high level language", which may be expected to be relevant to every level of the system proposed above.

The whole aim of the compiler writer is to design and implement a tool whose inner workings are irrelevant to the user. It merely takes a source code and produces object code. While the compiler is being used for small or straightforward tasks he is often successful, but the authors are not aware of any large scale embedded software program which has successfully avoided the need for at least one or two experts in the workings of the compiler being used. As both complexity and level of abstraction of compilers increase, this is not a situation which is likely to change. However, where compilers are intended for use as part of, or in conjunction with, 4GL systems, they are likely to be less complex (though possibly larger) than normal compilers.

5.1.6.2 Reliance on Database Management Systems

No reference has yet been made to database management systems, although Schussel, cited above, regards the relational DBMS as the first, and by implication the most important aspect of current technology to be mentioned. This may be because, as (ref. 5-6) has it,

> "(Scientific and military) users do complex things with trivial amounts of data, whereas most business computing consists of doing trivial things to large amounts of data".

In fact although it seems almost inevitable that the integration of our language system outlined above will rely heavily on database management technology, the user should not be forced to be aware of that fact.

5.2 CURRENT STATUS

5.2.1 Introduction

Just as in system building, it is often convenient to approach the design from the top down and to approach the implementation from the bottom up. So, by analogy, in Section 5.1 it was convenient to discuss the basic concepts of the language approach to automatic generation of control system software starting with the most general – the control engineering. In this section, the current status of components will be discussed starting with the most detailed aspects.

After discussing the language system of paragraph 5.1.5 "from the bottom up", this section will refer to some associated technologies which will contribute to the implementation.

5.2.2 Run Time Executives

The technology of run time executives for single computers suitable for embedded system use is well established. This is not to say that development is at end; issues such as run time efficiency, integrity and validation still cause concern, and leave room for optimization and improvement.

The problem of the run time executive for a real-time distributed computer network is much further from satisfactory solution.

These run time executive problems, however, although they may have some considerable influence on any successful automatic software generation system, are quite separable from its fundamental concerns, and will be given no further consideration in this chapter.

5.2.3 High Level Computer Languages

High level languages have received a considerable amount of attention in the last two decades. Cobol in the business community, Algol, Fortran and Pascal in the scientific and technical field, and Coral and RTL in the field of real time applications, have all achieved maturity and shown a remarkable degree of stability. The fact that there are more than 7.7 x 10^11 lines of Cobol in use, according to (ref. 5-3), may in part account for that stability.

It is in the highest degree unlikely that the control engineering community will have either the resources or the pressing need to develop a high level real time language tailored on it own purposes.

Ada and LTR3 are two languages released in the last few years to the real time software designer, and the investment in them will ensure that they retain pre-eminence in the field for many years.

The tools to support these languages, the APSE's and the IPSE's, are much less well established and the battle for supremacy will continue for some time. It is fortunate that considerable emphasis, in the form of public tool interfaces, has been given to ensure a wide and free market in these tool and toolkits. This should allow the access of global tools for configuration control or report writing to integrate the whole of the *language system* outlined in paragraph 5.1.5.

5.2.4 Software Engineering Languages

The process of expressing a software specification in a form suitable for automatic software generation has received a good deal of attention, and is currently showing some degree of success.

Balzer (ref. 5-7) gives a very concise summary of the elements of the problem.

> "... there are really two components of automatic programming: a fully automatic compiler and an interactive front-end which bridges the gap between a high-level specification and the capabilities of the automatic compiler. In addition, there is the issue of how the initial specification was derived. It has grown increasingly clear that writing such formal specifications is difficult and error prone. Even though these specifications are much simpler than their implementation, they are none the less sufficiently complex that several iterations are required to get them to match the user's intent. Supporting this acquisition of the specification constitutes the third and final component of the extended automatic programming problem."

Balzer's team developed a textual language, Gist, to capture software specifications. One particular noteworthy tool, developed to overcome the intransigence of a formal specification language, was the paraphraser. This takes a formal Gist specification as input, and produces a "natural" language paraphrases as output. Thus, Gist both summarizes, and provides an alternative point of view. It turned out to have a side benefit in that, although it clearly could not detect a disparity between specification and intent, it could, by virtue of providing an alternative viewpoint, make such a disparity much more obvious for human discovery.

Gist and its associated developments are academic, intended for discovery of truth rather than for development of industrial products. There is, however, a growing number of practical solutions to the software engineering language problem which are beginning to make commercial progress.

Examples of software engineering language include the following:

a. The work of Swann (op. cit.)

The language in which the software requirement is expressed is called (unhelpfully) LRM, standing for language reference manual. It is an extended form of JSD (Jackson System Development) with the addition of an implementation layer describing such things as process inversion and formal data types. The correspondence with JSD is sufficiently close that proprietary Jackson tools such as Speedbuilder can be used, and the language includes graphic elements.

A relational query language is available. The "compiler" is ASE (Application Support Engine) and the object code in Ada. The target is a large distributed network of computers, including some special purpose machines. This system is being used in earnest in a large and complex Submarine Command System.

In his conclusion, Swann says encouragingly

"Code generation is not pie in the sky, it is the logical consequence of modern software engineering and is usable here and now: given the necessary investment."

"The real requirement is nerve: to respond to a challenge with a new technology."

Clearly Swann's way needs the backing of a company with no shortage of this commodity.

b. Teamwork – (ref. 5-5)

Teamwork is a system based on the Yourdon/de Marco methodology, and interfacing with the Buhr Ada Structure graph notation. It contains some graphical elements, and encompasses tools for system design, analysis and modelling as well as automatic document production and Ada outline code generation. We are not aware of any current major project dependent on this method.

c. EPOS – (ref. 5-8)

EPOS also has extensive graphics support, as well as 3 specification languages for requirement definition, system specification and project management specification. It includes tools for management support, error diagnosis, documentation and communication and an Ada code generator. It is marketed in the UK, e.g., by Systematica, whose virtual software factory (VSF) workbench also claims Ada generation from several other input sources, including Mascot 3, SSADM, JSD, Yourdon and CORE.

None of the systems enumerated in a. through c. above makes reference to the interactive element which Balzer found unavoidable. Whether this is progress or reticence is not apparent from internal evidence.

A reference to translation into Ada code may imply partial translation only, i.e. the generation of an Ada framework and intercommunication information, will a high level functional outline structure. The ability to generate good quality Ada code is not typically available as yet except where restrictions at specification level make this practicable, as in more application oriented systems.

5.2.5 Application Oriented Languages

Software engineering languages are sometimes convenient for the representation of certain kinds of guidance and control applications. However, where the application embodies concepts which are more domain specific, the software engineering language needs to be supplemented by and integrated with an application oriented language.

It is hardly surprising that the control engineering software community has accumulated most of its experience so far in the automatic development of simulations rather than of operational flight systems. This fact is implicit rather than explicit in the published literature, and it is unfortunately therefore impossible to gather from that literature just how much software engineering effort has been needed to integrate control system software components generated by automatic means into flightworthy systems.

The examples that follow are divided into three classes, languages which are expressly intended to apply primarily to the design and simulation of control systems, languages in which the transition to an operational flight system has already been addressed, and languages which are specific to a particular subset of the flight system.

5.2.5.1 Design and Simulation Languages

a. ACSL

1. The Advanced Continuous Simulation Language, developed by Mitchell & Gauthier Associates (MGA) of Concord, Mass., is as its name implies targeted primarily at simulation.
2. It claims to be applicable to a wide range of simulation subjects ranging from biological systems to aerospace applications.
3. ACSL includes facilities to employ most of the usual control engineering methods for linear and nonlinear system analysis, and has been interfaced by a large number of other commercially available systems.

b. CTRL-C

1. This interactive computer aided design program has as its stated objective the provision of a "control system designer's workbench". It is primarily applicable to linear systems that can be represented in space-state form, and offers support for most of the design and analysis methods which fall within that limitation.
2. CTRL is based on the program MATLAB originally developed by C. Moler at the University of New Mexico.

5.2.5.2 Operational System Languages

a. MEAD

1. The Multidisciplinary Expert-Aided and Analysis Design Program (ref. 5-9), is a United States Air Force Dynamics Laboratory program that focuses on the development of a software tool for integrated control system design. Traditional design techniques and tools concentrate on the development of specific designs in each particular discipline. As these designs mature, they are

brought together and integrated as a system. At this time, many 'integration' problems are encountered. Often these problems are very difficult to fix and many times total redesigns are necessary, inefficient 'fixes' are required, or restrictions are placed on the final product.

2. The objective of MEAD is to develop a computer program that enables designers to assess the coupling effects between different disciplines early-on in the design process. This tool can be utilized by various contractors and government agencies to reduce control design development times, reduce system integration costs, and increase system performance. The initial MEAD program focuses on the integration of flight, propulsion, and structural control. It utilizes a sophisticated database management system to track and document the entire design process. Extensions to this program will include advances in control design methodologies and expansion of the expert-aiding capabilities. MEAD will reside on the Flight Dynamics Laboratory's VAX 8600 system.
3. The MEAD Computer Program (MCP) is being developed under the Multidisciplinary Expert-Aided Analysis and Design (MEAD) Project as a CAD environment in which integrated flight, propulsion, and structural control systems can be designed and analyzed. The MCP has several embedded computer-aided control engineering (CACE) packages, a user interface (UI), a supervisor, a data-base manager (DBM) and an expert system (ES). These modules have widely different interfaces and are written in several programming languages, so integrating them into a single comprehensive environment represents a significant achievement.
4. The MCP is a computer-aided control engineering environment for modelling, simulation, design, and analysis of linear and nonlinear airframes, engines, and structural models in state-space form. The CACE packages currently integrated into the MCP include the MATRIX-x (R) package for linear analysis and design, GENESIS, ALLFIT, and AUTOEXPEC (Flight-control-specific packages developed by Northrop Corp, Aircraft Division, Hawthorne CA 90250); the SIMNON program for nonlinear simulation, equilibrium determination, and linearization is being added at the present time. The MCP utilizes a supervisor that acts as the package integrator. All communications between the CACE packages, expert system (ES), data-base manager (DBM), and the user interface (UI) are coordinated by the supervisor.

b. MATRIX-x

1. This product, which is integrated into the MCP, was developed by Integrated Systems Inc. of Santa Clara, Cal.
2. It supports a very wide range of classical and modern design and analysis methods for linear and nonlinear systems, with a comprehensive range of simulation aids and good graphic output facilities.
3. Included in the MATRIX suite is the AutoCode automatic code generation system described in c. below, and a rather significant aid to verification and perhaps a certification is provided by the facility link the automatically generated software back into the simulation package in order to verify the integrity of the generated code.

c. AutoCode (AuCo)

1. Autocode is a graphical programming environment consistent with the Ward/Mellor and Hatley/Boeing real-time methodologies. It is explicitly aimed at the automatic generation of operational

control system software. It claims to 'generate error-free, fully documented C, Ada or Fortran coded in minutes'.

2. (AuCo) makes the more ambitious claims that software documentation is required only at systems level, and that software requirements, software design, and software module testing are all rendered unnecessary.
3. Applications already made include AutoCode generated Jovial used on the production A6 SAFCS, AutoCode generated 1750A code used on the F15 STOL, and 12 complete sets of control laws flown experimentally on the F15.

5.2.5.3 Special Subsystem Languages

FISP is one example and is representative of a number of others. FISP is a system, developed by GEC Avionics, Rochester, which provides a means of rapid prototyping; and automatic implementation of Head up Display (HUD) symbology. FISP stands for Flight Instrumentation Standards Program, and it comprises two parts, a Symbology Development Workstation (SDW) and a compatible airborne display computer. FISP is currently under evaluation at Wright-Patterson AFB. The SDW provides for the user of the facility to develop and experiment with new symbology. When the set is complete a standard ground program loader (GPL) can be used to transfer the symbology set directly from the DSW to the ADC.

5.2.6 Associated Technologies

Reference has been made several times in the foregoing to the desirability and to the availability of graphical representation in both the control and the software engineering domains. The technology for handling information in graphical form is clearly well advanced, with high resolution screens, mice, windows, icons and pointers all readily available.

A view has been expressed that there is always a necessity for an equivalent textual representation, but this is debatable. It is always possible to provide a textual equivalent to any graphical representation, but a simple and elegant graphic may permit only a very clumsy translation into text, and in that case the value of the text seems questionable. One picture is (sometimes) worth a thousand words.

Other representation technologies may well have application. For example, state transition matrices may be supported effectively by spreadsheet interface concepts derived from quite different applications. Such techniques may be useful in the manipulation of state vectors and of other multidimension control engineering concepts.

Tools for handling the highest level specification, such as requirement animation or rapid prototyping may find a place in the scheme.

5.3 APPLICABILITY OF THE LANGUAGE APPROACH

5.3.1 Prerequisites for Success

The latter part of Section 5.1 above outlined where we would like to be. Section 5.2 contrasts that with where we are now. The difference between the two represents a very considerable investment of time, effort and money. This investment will only be forthcoming if there is good prospect of a return, and that return will be worthwhile only if:

a. The application field is sufficiently stable that the primitives and constructs of the domain specific language will hold currency for an appreciable period.

b. The number and scale of application within the domain in that period are sufficient.

5.3.2 Applicability to Types of Guidance and Control Software

Table 5-1 indicates the applicability of the 4GL approach to the types of software under consideration stated in figure 1-1.

Without denying the possibility of applying a 4GL approach to some of the non-operational types of software in table 5-1, e.g. the use of compiler compilers within the category Software Design and Implementation Software, or the use of ATLAS as a 4GL approach to the provision of ATE Software, this report primarily considers the applicability of the 4GL approach to Guidance and Control Operational Software, (i.e. Operational Flight Program Software and ground-based Mission Planning Software). In addressing these application domains the effect of the 4GL approach on Requirements Capture, System Design, etc., will be addressed.

Type of guidance and control related software	Applicability
Requirements capture and documentation software	N
System design	
• Concept formulation and development	N
• Design	A
• Design capture and design documentation	N
System simulation	
• Vehicle simulation	N
• Navigation, guidance, and fire control simulation	N
• Pilot-vehicle interface simulation	N
• Mission execution simulation	N
Operational flight program	
• Flight (safety) critical	E
• Mission critical	E
• Run-time environment	N
• Operating system/executive	N
• Controls and displays	E
Mission planning	A
Software design and implementation	E
Management support	A
G&C system V&V	N
G&C system certification	N
ATE	E

Key:
E = existing example of 4GL generator known in ths domain
A = applicable
N = no apparent incentive for creating 4GL generator

Note:
This table addresses the question of whether the designated type of software (e.g., management support software) could be usefully specified, designed, generated, tested, etc. by a 4GL approach, not whether a 4GL approach to the generation of operational flight program software could provide management support information. This latter question is addressed in section 5.2.4.

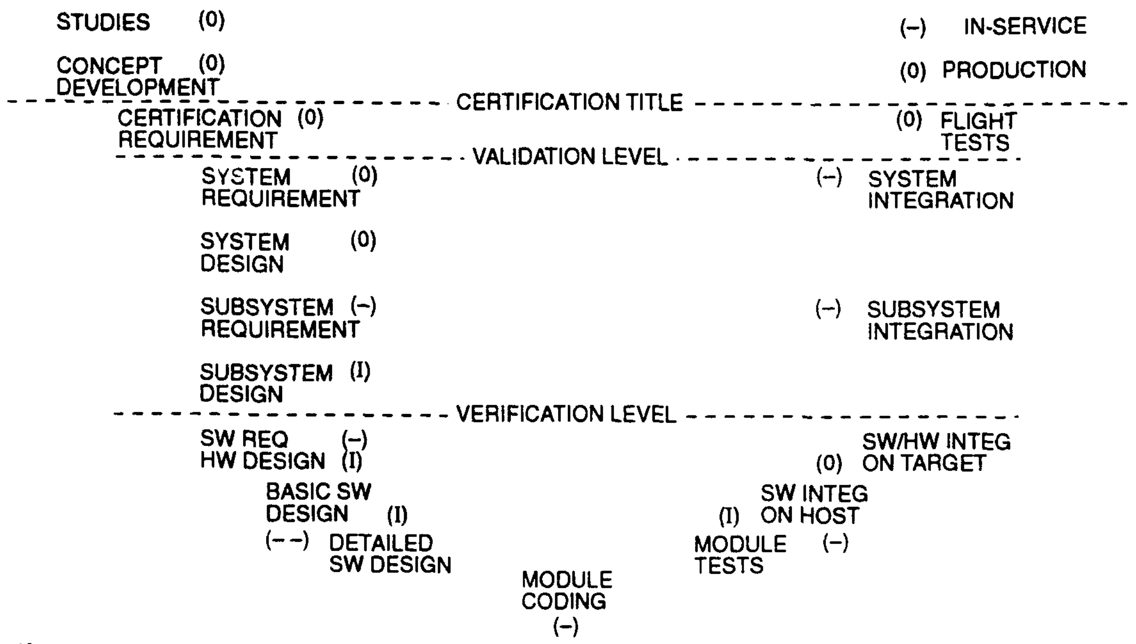
Table 5-1. Applicability of the 4GL Approach to the Generation of Different Types of Guidance and Control Software

5.3.3 Applicability to the Life Cycle

Figure 5-6 shows the system/software stage representation (V-diagram) in the form presented in [GCP90] with annotations to show the effects of a 4GL approach to the Operational Software. The diagram is, of course, based on the traditional approach to software development, and is compatible with an IPSE approach since that is by definition an integrated support for the same approach, but is shown here for comparison purposes.

As a frame of reference for considering the various kinds of activities involved and to illustrate the meaning of terms like 'system' and 'subsystem', it may be helpful to consider the following two cases:

a.	System:		Vehicle Management System (VMS)
	1.	Subsystem:	Flight Control System (FCS)
	2.	Related subsystems:	Flight Management System, etc.
	3.	Subsystem function:	flight critical inner loop control of aircraft
	4.	Subsystem hardware:	combinations of 2 dissimilar processor types
b.	System:		Integrated Displays System (IDS)
	1.	Subsystem:	Computer Symbol Generator (CSG)
	2.	Related subsystems:	Primary Flight Displays System, Display Heads, etc.
	3.	Subsystem function:	generation of primitive graphic commands and driving from application-oriented displays requirements and parameters
	4.	Subsystem hardware:	single general purpose processor controlling displays hardware



Key:
Using a 4GL approach the effort involved in the stage could be increased (+), decreased (–), or greatly decreased (– –) compared with a mainstream system/software development approach. (I) = after initial investment effort is less. (0) = no significant change. See section 5.2.4.1 for explanatory notes.

Figure 5-6. V-diagram Software Development Components Potentially influenced by a 4GL approach to Software Development

The potential impact of 4GL approach on life cycle stages are below:
Studies (e.g., of VMS or IDS trade-offs with other major systems such as wing characterization).

Little impact, except that in the course of undertaking such trade-offs it may be useful to undertake rapid prototyping/simulation to examine potential behavior in various flight regimes. This could be facilitated by the rapid prototyping aspects of the 4GL approach.

Concept development (e.g. of VMS or IDS internal architecture, e.g., that resulting from a change of wing design) - no change.

Certification requirement (e.g. of VMS or IDS certification requirements) - No change at the system level, but if investment has been made in a 4GL system the details of the software aspects of the certification process will change.

System requirement - no change.

System design - no change.

Subsystem requirement (e.g. of an FCS or CSG) - no change.

Subsystem design (e.g. of an FCS or CSG) - This is the point at which a 4GL approach might be expected to start having major significance. A 4GL approach could assist in control law design. In order to be applicable, the 4GL system must be able to embrace all options of single/multi processors, processor type, input-output, etc. which may be selected here.

Software requirements (e.g., of an FCS or CSG) - A 4GL system will have narrowed the options by supporting only certain executive or RTE structures and certain software languages with which to interface. The mapping from subsystem requirement to (subsystem) software requirement will require less effort in that part of the software that is application domain oriented, e.g., in the mapping of control laws into software requirements. Indeed it will be possible to bypass this stage for that part of the software.

The specification of the software documentation structure will be simplified, but where the customer or procurement agency has specific requirements, a cross-correlation will need to be made and exceptions justified.

Hardware design (e.g., of an FCS or CSG) - A 4GL system will have narrowed the hardware design options.

Basic software design (e.g., identifying the software modules within the real-time structure, specifying inter-module interfaces, etc.) - A 4GL system will reduce the modularization and characterization of interfaces.

Detailed software design (e.g., specifying the functions of a software module algorithmically, designing the internal trade-offs, specifying the implementation in PDL or otherwise) - A 4GL system will lessen the effort, and in parts by-pass this stage.

Module coding (e.g, coding a module from, say, a PDL) - Significant reduction in effort and documentation. The more difficult parts of the system will be resistant to automation, e.g., BIT and RTE. The difficulties that arise from lack of human consistency, however, will be dramatically minimized.

Module testing (e.g., testing low-level modules stand-alone, and higher level modules using stubs and test harnesses, generating and recording sufficient test cases, predicting their results, fanning out into a system of partially adequate and co-existing modules requiring review and configuration management) - A 4GL system should lessen the number of errors and may eliminate this stage for some types of module.

Software integration on host (e.g., building and testing of integrated combinations of modules) - A 4GL system should lessen the time taken to clear problem reports, by virtue of reducing the number of levels of software and the number of items.

Software/Hardware integration on target - No change, except by virtue of possibly fewer errors surviving to this stage.

Subsystem integration - Inevitable changes of requirement appreciated at this stage may be incorporated more rapidly using a 4GL approach, allowing faster integration.

System integration - A 4GL may be expected to simplify changes and system builds.

Flight tests - A 4GL may be expected to simplify changes and system builds.

Production - No change.

In-service - Possibly benefits will be achieved in customer satisfaction by ability to change the requirements and re-introduce the software, but this is probably a long way off.

5.3.4 General Aspects of Applicability

In order to be acceptable a 4GL approach must be able to handle:

a. The requisite size and complexity of control system. Demands are likely to go on increasing.

b. The complexity of the target system. Distributed targets are likely to become more common, but not all the problems have yet been solved satisfactorily even with human intelligence applied directly to them. To apply intelligence indirectly to the problem through automated aids will be harder.

c. The degree of integrity needed. The use of automatically generated software has a complex effect on system integrity – one class of errors to which humans are prone is removed but a new class, generally of the form of the results of exercising the automatic system outside the envelope in which it is valid, is introduced.

5.3.5 Applicability to the Support of Systems and Software Engineering Viewpoints

5.3.5.1 Applicability to Requirements Development

To facilitate the capture of a control system requirement and the subsequent development process the 4GL system must offer interactive communication with the control engineer in the terms of each of the design methods which are necessary for the design. System responses in temporal and frequency domains and various appropriate stability criteria can be progressively tailored as the system specification evolves.

With increasing use of digital controllers in control loops, more complex non-linearities can be included in the design, giving more favorable performance at the expense of more difficulty in system analysis.

5.3.5.2 Code Generation

The efficiency of code in an inner control loop is a parameter of first order importance in the design of a digital control system. It will be advantageous to provide the 4GL system with the facility to provide a high level of optimization of running time for particular subsets of code.

5.3.5.3 Configuration Management

Insofar as a 4GL approach reduces the number of levels of requirements, specifications, design, testing, etc., the configuration management task will be inherently reduced. However, unlike in rapid prototyping, a 4GL approach used for mainstream guidance and control systems would need significant human involvement in configuration management. Choices have to be made for associations of components in various states of change, and explanations given for component status in relation to baselines, and reasons and implications of change noted. These are not amenable to automation in a 4GL system or any other.

A configuration management system which deals only with software will not be adequate for system design, and the 4GL system must provide the ability to log and track design changes in sensors, actuators, computers and any other system components in conjunction with the configuration management of software modules.

5.3.5.4 Documentation

Control systems have special documentation needs particularly in the graphical expression of control system properties.

5.3.5.5 V&V and Certification

There is room for automation in the design of development testing and with the selection and measurement of trials results and their comparison with theoretical prediction.

5.3.5.6 Phased Development

Real development programs do not usually start with a fully worked out requirement specification and proceed in a single smooth flow to a fully satisfactory end product. Development programs generally proceed by phases, whether those phases are pre-planned or not. The 4GL must allow for this fact.

5.3.5.7 Maintenance

The requisites for design maintenance, as opposed to field servicing, are very similar in nature to those of phased development discussed above.

5.3.5.8 Project Management

Project management should be inherently simpler in any unified system of support, such as a 4GL system, provided that the system is well-designed and well-understood by its users. Teething troubles may be significant. A 4GL system will have higher granularity of visible components, and this will lessen the quantity of project management documentation.

5.4 LIMITATIONS

5.4.1 Dependence on Specific Hardware

The user interface with the system is likely to be a proprietary workstation. If the intermediate level is a standard software engineering "language" then support of target types will depend on that implementation.

In general there should be no exceptional problem in maintaining any one component of the system. The real problem is likely to be that there are so many components, all of which will require maintenance from time to time, that keeping them all compatible will be difficult. It seems likely that the system will require a maintenance tool to aid the retention of compatibility.

5.4.2 Extensibility

The fact that interactive bridging – the equivalent of code inserts – will probably be necessary at a number of levels in the system has already been discussed. This renders more difficult most of the integrating features of the system, e.g. configuration control, or validation of the output from end to end.

5.4.3 Penalty in Run Time Performance

It is not possible to estimate the overall efficiency of such a system, which would depend enormously on the details of implementation. It seems likely that there would be some penalty, in as much as a dedicated human "tuner" could always provide some speeding up on run time performance.

5.4.4 General Limitations

The 4GL approach (or any other thoroughgoing automatic approach) to automatic software generation is likely to be inapplicable or not cost effective in areas where the realization of the software must be achieved under a number of quite commonly occurring constraints.

Such constraints include:

a. The use of innovative hardware,

b. Tight performance or memory constraints,

c. Intrusive system considerations (e.g., unusual certification requirements),

d. Changes in customer's support requirements.

5.5 FUTURE R&D

The 4GL approach to the automatic generation of control system software has been presented. Most of the components of such a system exist, but while some are already mature others are only embryonic. Some complete systems with limited flexibility in their hardware and software targets have been built.

The economics and worthwhileness of 4GL systems depend very heavily indeed on stability of application domain and hardware, and on the size of the software being large enough to warrant the initial investment. The maintenance of 4GL systems in a way that will give them resilience to the inevitable changes in the application domain is not a trivial task and should not be underestimated.

5.6 REFERENCES

5-1 McNicholl, An overview of the Paper on CAMP program, presented to GCP/WG-10, Oct. 1988.

5-2 Reed, Going Fourth, Informatics, Sep. 1987.

5-3 Schussel, Software's Future, Computer Systems, Jan. 1988.

5-4 Dijkstra, E. W., Notes on Structured Programming, published in O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press.

5-5 Wachsmuth, Teamwork Documentation, sent to GCP/WG-10 members, Nov. 22, 1988.

5-6 Barry et al, The Automatic Generation of Computer Graphics Source Code: No Programming Experience Necessary, IEEE NAECON 88.

5-7 Balzer, A 15 Year Perspective on Automatic Programming, IEEE Trans S/W Eng, SE11(11), Nov. 1985.

5-8 EPOS Overview, Systematica

5-9 Hummel, Taylor, Multidisciplinary Expert-Aided Analysis and Design (MEAD), Proc. 3rd Annual Conf. on Aerospace Computational Control, Oxnard, CA, Aug. 1989.

BIBLIOGRAPHY

[Pri 88] Primrose, Staying with Cobol, Computer Systems, Jan. 1988.

[GCP 90] AGARD Advisory Report No. 229, The Implications of Using Integrated Software Support Environment (sic) for Design of Guidance and Control Systems Software, Ed. Ed B. Stear.

[Cle 88] Cleveland, Building Application Generators, IEEE Soft, July 1988.

[Swa 88] Swann, Code Generation for a Large Command System, Paper presented to GCP/WG-10, Apr. 1988.

CHAPTER 6

COMBINATION APPROACH TO SOFTWARE GENERATION

6.1 OVERVIEW

Obviously, the four approaches to software generation investigated by Working Group 10 will not be used in a completely self-supporting and independent manner. For example, a fourth generation language may be used to generate the requirements for a guidance and control (G&C) system, but the progressive refinements of the requirements specification may be accomplished by transforming the specification or a subset of the specification into a formal language and repeatedly applying formal transformations to generate an executable form of the specification. In all phases of the transformation, expert system technology can by used to aid personnel in accomplishing and monitoring the transformations. Further, reuse technology can be useful at all levels of the transformations: reusable portions of system requirements from G&C systems from similar existing aircraft, reusable portions of similar system and subsystem design, reusable software modules and reusable test plans. Access and necessary modifications to these reusable objects can be provided by expert systems.

Thus, any useful, comprehensive software generation system will probably combine portions of all four technologies. In this chapter we examine some of the obvious combinations and the interactions among the component technologies.

6.2 EXPERT SYSTEMS APPROACH WITH OTHER APPROACHES

The expert system approach can complement each of the other three approaches, and it is applicable in theory to almost every phase of the software generation process. For instance, an expert system can aid a person using a fourth generation language (4GL) in producing a specification by checking for contradictions, redundancies, and completeness in the resulting specifications; it can identify constructs generated in the 4GL that are likely to lead to inefficient or obtuse code; and it can be used to access reusable components through the process. It should be noted that the current state of development of expert systems may not be mature enough to be useful in all these applications. However, the potential exists for expert systems to be useful to some degree in all phases of software generation, so their use is implicit in all the combinations of the other three approaches.

6.3 FOURTH GENERATION LANGUAGE WITH OTHER APPROACHES

An interesting combination is the use of a fourth generation language specifically designed for G&C system generation supported by extensive libraries of reusable objects. The ultimate goal for such a combined system would be the generation of executable code based on the system specification entered via the 4GL. Given the existing state of the technologies, many intermediate steps between the 4GL description and executable code would be required. Even when technology improves to the point that such a one-step translation is feasible, it may still be desirable and more efficient to have a number of intermediate steps.

Both the 4GL and the reuse system should be specifically restricted to the guidance and control systems domain; the 4GL should have language constructs specific to specifying and programming G&C systems, and the reuse library and retrieval system should have modules and functions specific to G&C systems. A G&C requirements expert then uses a 4GL to describe the requirements for a G&C system for a new aircraft. The resulting specification generated would comprise several parts: a tabular specification of a subset of the requirements, sets of specifications for sequences of actions (normally in some lower level language), specifications of timing requirements, and identification of possible useful subsets from existing G&C requirement specifications. Such a 4GL would be designed so that certain language constructs are recognized that generate queries to the reuse system. At this level, the reusable objects are parts of specifications. Candidate subsets would be delivered to the 4GL processor, and these subsets would be modified as needed and integrated into the new specification.

At this point in the software generation process, the combination of the 4GL and reuse system has produced a specification for the new system. This specification would then be given additional checks (both manual and automated) for consistency and completeness. The resulting specification details *what* is to be done without indicating *how* the specifications are to be implemented. The next step is to introduce the detail needed to specify in some lower-level notation, *how* the specifications are to be implemented. Lower level translators within the G&C 4GL processing system would then translate the specification into a lower level intermediate language (one closer to executable code). Again, the 4GL translator calls upon the reuse library system for applicable modules. Candidate modules are returned to the 4GL processor to be modified as needed and integrated into the software design.

It is important to note that at this and lower levels in the software generation process, the reuse library must store timing parameters for the modules as well as parameterized functional descriptions. Such estimates of timing parameters must necessarily depend on assumptions concerning implementation details.

The next step for the G&C 4GL processing system is the translation of this software design from the intermediate language into a detailed software design by generating code for the software modules. Again, at this level, the G&C 4GL would access the reuse library for existing software modules that are modified as needed and integrated with the other modules generated by the 4GL translator.

This point in the software generation process is analogous to that of the individual module testing and system integration in the normal software development life cycle. Although testing and integration would still need to be done for the modules generated by the G&C 4GL system, the testing would be substantially reduced because of the reuse of many already-tested specifications and modules.

The G&C 4GL system ideally would produce test plans for the software by accessing the reuse library's file of existing test plans. Again, the plans would be modified to fit the new application.

This combination approach is only one way a G&C 4GL system and a specialized G&C reuse system might be used in the software generation process. In an idealized 4GL system, the processor might access the reuse system only at the top level to retrieve subsets of specifications. Then, the 4GL system would interactively translate these specifications into executable code. But, from a practical standpoint, such a powerful translation system is not achievable in the near future. A successful G&C 4GL system is likely to access a reuse library at each level of the translation process in order to generate code that can meet the timing and efficiency requirements of G&C systems.

CHAPTER 7

CONCLUSIONS

7.1 GENERAL OBSERVATIONS

There have been many tools devised over the past decade for reducing the high cost of software development. These tools have followed a normal pattern of tool evolution, similar to the tools developed for digital hardware design. These tools range in development phase applications from requirements capture to logistic support, and they range in scope from design and analysis to system level CASE environments. With the introduction of each new software development tool and method, incremental gains have been experienced in productivity and software quality.

One of the tasks of Working Group 10 was to survey the current status of software development tools for guidance and control applications. The results of this investigation were gratifying; much work has been performed within the aerospace community on these development tools with encouraging results. Within the software community in general, software generation tools and methods have greatly improved over the past few years. One of the most notable programs that is making great progress in tool and method development is the European ESPRIT program. The United States STARS program is making good progress as well. In some cases, tools developed in these programs and other software initiative projects have been tailored for use in the guidance and control domain with excellent results.

It was convenient for study purposes to divide the methods for generating software into four approaches: (1) reusable software, (2) knowledge-based support and automation, (3) program transformation (the use of formal specifications), and (4) fourth-generation language. In actual practice, however, these approaches do not stand alone; software development tools usually employ combinations of two or more of them.

There are a number of barriers that must be overcome before the software generation techniques can be used effectively to significantly increase productivity. First, it must be understood that it will be necessary to make a substantial investment in time and manpower before large increases in productivity will be achieved. Furthermore, there must be a change in attitude toward using proposed new methods to develop software; there appears to be a mindset that members of software development teams develop early in their careers that resists the adoption of new techniques. This attitude can only be changed by a commitment from management to use and enforce proven new methods. These new methods must also be introduced into the educational process, so that each year the new software developers are primed to use these concepts.

Each one of the four software generation methods investigated by Working Group 10 can increase software development productivity in guidance and control applications. However, the increase that can be expected from each of the four methods taken individually will not achieve the "order-of-magnitude" improvement many have wanted or expected. Working Group 10 agrees with Frederick P. Brooks, Jr., who stated in his "No Silver Bullet" article that "...as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity."

7.2 UTILITY TO GUIDANCE AND CONTROL SYSTEMS OF THE SOFTWARE GENERATION METHODS STUDIED IN THIS REPORT

The four approaches to software generation described in chapters 2, 3, 4, and 5 of this report have very different objectives, have different impacts on the structure of the software development teams, and are applicable to different parts of the traditional software development cycle.

The reusable software approach to software generation aims to minimize the software development effort by using existing software elements in new systems. Analysis of this approach has shown that development cycle costs can be reduced by 25% to 45%, and total life cycle costs can show an improvement of 11% to 26%.

There are a number of reusable software concents being used in guidance and control applications, the most notable one being the CAMP program in the United States. The CAMP program (and a few others) have proven that the reuse concept can be applied effectively across many phases of the design process and that development costs can be significantly reduced in guidance and control systems. But it has also been proven that the reuse methodology must be built into the development process before a project is started; reuse cannot be added as an afterthought and be expected to reduce development costs or schedules. At the beginning of the development phase, certain mechanisms must be in place to fully exploit the software reuse concept. First, there must be some sort of a library system for storing and retrieving reusable objects. Second, there must be a mechanism for giving the system designer all the information he needs to fully understand the functions and interfaces of the stored software object. Third, tools for tailoring the reused object to the new application must be available. Fourth, tools for integrating the objects into the new system are required.

Unlike the reuse concept which produces productivity gains by reducing or eliminating certain specific development tasks, the knowledge-based approach captures and provides expert system help at all points of the development cycle. Knowledge-based (or expert) systems can be applied in the development process to:

a. Provide functions such as tutoring, user guidance, documentation control, and information retrieval over all phases of the developmental process.

b. Identify and retrieve reusable software components, such as pattern matching and deciphering semantic descriptions. Examples of this concept can be found in the PRACTITIONER (an ESPRIT project) and the CAMP projects.

c. Assist in the transformation process when applying formal methods. Examples of this concept can be found in the STES and REFINE projects.

d. Be a part of the software generation and automated programming function itself.

Knowledge-based approaches are normally applicable to specific domains, and current developments in support tools for software generation show that this concept can be used effectively in guidance and control systems. An example is the Air Force MEAD (Multidisciplinary Expert-aided Analysis Design) program, which is developing a software tool for integrated central system design.

The program transformation approach is a method that is being aggressively pursued to solve the serious recurring problem of how to effectively and precisely transform incomplete system requirements into system software code. Because the transformation rules used in this method are a formal mapping between the requirements and the produced software code, the functionality of the software code is both preserved and mathematically traceable. Due to the strict mathematical principles used in transformational programming, it is attractive as a potential tool in developing automatic software code generators.

One of the most significant ongoing system projects using transformational programming is a project sponsored by the ESPRIT program. Initiatives within this software technology project, such as the PROSPECTRA, TOOLUSE, RAISE, and REPLAY, emphasize transformational programming. Other initiatives, such as the SAFE and TI programs at the Information Science Institute of the University of Southern California; and LIBRA, PMB, PECOS at Edinburgh are investigating the use of formal specifications. IBM Hursley Laboratory, Bull in France, ONERA/CERT/DERI, and Lockheed are experimenting with program transformations as well.

As stated in chapter 5, the term "fourth-generation language" (4GL) was invented to label a growing number of products which enable a computer to be programmed by the designer who needs to specify "what" is to be done in his own domain-specific language, and then leaves the "how" it is to be done to the 4GL. The 4GL approach can be thought of as a combination and evolution of the rapid prototyping and the integrated program support system concepts.

The 4GL approach uses a specific, application domain language made up with all the primitives, constructs, and syntax required by the language. This language is then used by the design specialist to communicate with the software development tools at all phases of the development process.

4GL systems have been developed for software engineering and other disciplines as well. Within the guidance and control domain, some commercial programs have been developed, such as ACSL, CTRL, and MATRIX$_X$.

7.3 FUTURE RESEARCH REQUIREMENTS

Throughout the study of the software generation approaches for guidance and control applications, there was the recurring concern about flight safety issues. In the software reuse approach, for example, the reused software must go through the same verification and validation phase as newly developed software, since the system configuration and operating parameters are, by definition, different. The use of expert and artificial intelligence systems in the software development approach is particularly troublesome, since these systems do not always produce mathematically predictable and consistent results.

The use of formal specifications (transformation programming) offers one potential solution to the problem of producing safe, flight critical software. One of the major problems with transformation programming, however, is that it uses mathematical methods not normally taught to guidance and control and other engineers. As a result, there is a breakdown between the guidance and control engineer who establishes the design requirements and the mathematician who converts the requirements into transformational rules. Further research is required to develop improved communication methods between the guidance and control engineer and the transformational program process. There are no major technological breakthroughs required to make the software reuse approach work in guidance and control applications. The major limiting factor in this approach is the lack of well-designed libraries for reusable software. A second limiting factor, and possibly the most important one, is how to handle the business aspects of reusable software. These issues include questions of who will bear the first-time development costs, and who will bear the liabilities in terms of performance, safety, and maintenance. Development of the reusable software approach should follow and build on the CAMP program.

It is believed that a software development system for guidance and control systems built using a combination of the four approaches will provide the greatest payoff. Chapter 6 provided some insight into how these combinations might effectively proceed. However, the combined approach must be carefully planned and must start with documented requirements and design specifications that have been developed and agreed on by the guidance and control system community.

Most of the technology and components for a 4GL software reuse and expert system exists, but many of the components are in the embryonic stage. There are no major technological reasons why a full automatic software generator for guidance and control software cannot be developed using combinations of the four approaches considered in this study. However, the future depends upon the willingness of the community to make the investment necessary to produce the system.

7.4 REFERENCE

Brooks, Jr., Frederick P., "Essence and Accidents of Software Engineering", Information Processing, 1986.

APPENDIX A

AUTOMATIC SOFTWARE GENERATOR DEVELOPMENT ISSUES

A.1 DESIGN METHODS

Any software generator must be based on specific design methods. Therefore, the best software design methods for a particular application must be selected before a software generator can be selected. Also, different approaches to software generation suit different design methods.

A.2 REQUIREMENTS CAPTURE

To implement software generators, requirements must be captured in a machine-processable form so that they can be used as inputs to the generators. This is likely to place some constraints on requirements capture methods and techniques, depending on the particular software generation approach used.

A.3 ROLE AND USE OF PROTOTYPING

Prototyping is becoming more widely used as a technique to refine and validate requirements. Software generators probably have a substantial role in software prototyping, even if they cannot achieve the overall efficiency and performance required of the real system. Particularly important is the expectation that software generators will be able to generate prototype software much faster than could otherwise be done. This would minimize the effect of prototyping on the overall design and implementation schedule and might also enable repeated use of prototypes for a high level of functional requirements refinement.

A.4 TASK ASSIGNMENT IN REDUNDANT FAULT TOLERANT MULTIPROCESSOR ARCHITECTURES

As noted previously, modern navigation, guidance, and control systems are typically implemented using redundant, fault tolerant, heterogeneous multiprocessor architectures. If software generators for these systems are to be effective, they must be able to assign tasks on such architectures. Again, this will require a good model of the fault tolerant and reconfiguration properties and the timing performance of software tasks on such architectures.

A.5 NECESSITY AND STRUCTURE OF ALTERNATE VIEWPOINTS

There are different viewpoints that are important for any particular piece of software and this is true for navigation, guidance, and control software. Obviously, any software produced by software generators must provide for such viewpoints.

The buyer's viewpoint is typically represented by the system requirements and specification documents together with acceptance test procedures. The former represents inputs to software generators, while the latter should be represented as generator outputs.

The user's viewpoint is represented by descriptions of the functionality of the software and appropriate documentation describing its applications, limitations, installation, error handling, error message structure, etc.

There are certain viewpoints which are especially important in software design and implementation. These viewpoints include the following:

a. The functional viewpoint is represented by a functional description of the various modules of a piece of software.

b. The data flow viewpoint is represented by a data flow graph and represents the flow of data through the software at time of execution.

c. The control flow viewpoint is represented as control and execution of the program, including control of the data flow.

d. The static structure viewpoint is represented by the structure of individual modules and their inner connections.

e. The dynamic structure viewpoint is a representation of the dynamic structure of the software at the time of execution.

There are several other viewpoints which are important during software integration and test:

a. The test viewpoint is represented by test specifications which prescribe the nature of the test to assure its proper functionality and operation, including the input stimuli to be used and the output response to be recorded and checked for correctness.

b. The verification and validation viewpoint is a check to see that the software modules and the software programs satisfy their specifications and meet the system requirements.

c. The certification viewpoint is represented by documents which describe the process to be used by Federal authorities to certify that the software meets prescribed levels of flight safety.

d. The quality assurance viewpoint is represented by documents describing procedures used to assure the quality of the software. For the most part, these documents describe the procedures and tools used to develop the software, and the quality assurance function is verifying that the actual procedures and tools used are consistent with those described for the project.

These test and integration viewpoints place certain requirements on software generators for the necessary hooks into the software generated to be able to perform the tests and to provide appropriate documentation.

The configuration and control viewpoint is represented by configuration and control requirements placed on the software generator that the generator must be able to incorporate and support.

The run-time environment resource management viewpoint can vary widely from system to system. It is represented by the run-time environment requirements and the real-time management of the run-time environment. Obviously, software generators must be able to accommodate run-time environments and their related resource management.

A.6 FUNCTIONAL VERSUS OBJECT-ORIENTED DESIGN

Some software generation approaches will lend themselves more to functional design, others will lend themselves more to object-oriented design, and still others will lend themselves to other kinds of design techniques. This is a significant issue that must be explored relative to software generators.

A.7 ASSISTED VERSUS AUTOMATIC GENERATION

There is a split in the software community as to whether software for design and implementation should assist programmers or should generate software automatically. There are arguments pro and con. It seems obvious that effective automatic generation is preferred, but this is a worthwhile issue to explore.

A.8 INTERNAL COMMUNICATION MECHANISMS

Various internal communication mechanisms have been used between modules of software programs for I/O. Again, determination of the kinds of communication mechanisms that best lend themselves to different approaches to software generation is an open issue and must be addressed.

A.9 ROLE AND USE OF INTERMEDIATE LANGUAGES

The role and use of intermediate languages in software generators will vary depending on the approach used for software generation. Some approaches may use a large number of intermediate languages, while others may use only a few. Based on experience with high-level languages and compilers, it is unlikely that effective software generators can be implemented that do not use intermediate languages.

A.10 PROCEDURAL VERSUS NONPROCEDURAL LANGUAGES

Clearly, any approach to software generation requires nonprocedural languages to express software requirements and specifications. It also seems inevitable that any approach to software generation use one or more procedural languages for the how-to part of the software being generated. The proper number, types, and characteristics of both nonprocedural and procedural languages for different approaches to software generation is an open question which needs to be explored.

A.11 SYNTAX HANDLING

Software generators must receive inputs from higher level design processes and must provide output for the test and integration process. It is likely that both the inputs and outputs will use a wide variety of text and graphics languages and their related syntax. To maximize flexibility, it is important that software generators be able to handle a wide variety of syntax and automatically transfer from one form to the other.

A.12 SEMANTICS HANDLING

As noted earlier, software generators, to be truly powerful, must be directly usable by systems engineers on actual applications. Accordingly, software generators must be able to understand the semantics of the applications area for which they are being designed.

A.13 MODIFICATION SUPPORTABILITY/EASE OF APPLICATIONS AND SUPPORT SOFTWARE

As noted earlier, guidance, navigation, and control software is frequently updated during its lifetime and much of the associated support software may also require frequent updates. Therefore, it is important that any tools for design and implementation of navigation, guidance, and control support modification of the resulting software. In principle, this should be easy for software generators because modifications could be made by merely changing the software requirements/specifications and invoking the generator.

A.14 SUPPORT ENVIRONMENT REQUIREMENTS

Once developed, software generators will be tools, albeit powerful tools, just like other tools for the software design and implementation process. As such, they should be designed to be easily integrated into modem software engineering environments.

A.15 DOCUMENTATION

Proper documentation is essential to the test, integration, postdevelopment suppcrt, and operation of all applications software. As much as possible, software generators should automatically provide documentation for the software that they generate.

A.16 CONFIGURATION MANAGEMENT AND CONTROL

All applications software and the key documents used in its design and implementation have to be put under strict configuration management and control. Because software generators will support modifications to the software, strict configuration management and control of software generators will also be mandatory.

A.17 GENERATOR INTEGRITY

Because generators automatically produce applications software directly from software requirements and specifications, the integrity of the generators must be comparable to the software integrity. Otherwise, a significant amount of testing, verification, and validation of the resulting software will be required.

A.18 ASYNCHRONOUS VERSUS SYNCHRONOUS SYSTEMS

Much has been said about asynchronous versus synchronous operation in navigation, guidance, and control software. Since there are sound arguments for including both operations in many systems, software generators should be able to accommodate both asynchronous and synchronous operations, individually and in mixtures.

A.19 CONCURRENCY

Concurrency is essential in redundant, fault tolerant, heterogeneous multiprocessor architectures typical of modem navigation, guidance, and control systems. Hence, software generators for such systems must be able to efficiently implement concurrency.

A.20 DETERMINISTIC VERSUS NONDETERMINISTIC PROGRAMS

Hard real-time constraints associated with guidance, navigation, and control systems will require either deterministic programs or nondeterministic programs with tightly bounded execution time. Any satisfactory approach to software generation must deal effectively with deterministic and tightly bounded nondeterministic programs. However, there may also be modules within programs which are just nondeterministic and they should not be excluded by the use of software generators.

A.21 RUN-TIME ENVIRONMENT: DEFINITION, DESIGN, AND DESCRIPTION FOR GENERATORS

For software generators to produce executable software, a precise description of the run-time environment must be provided as input to the generators. Also, for generators to have wide applicability, they should be able to accept a wide variety of run-time environments and to adapt their functioning to generate software compatible with these environments.

A.22 EXTERNAL COMMUNICATION MECHANISMS

Navigation, guidance, and control software must communicate with a wide variety of input and output devices. For this, appropriate external communication mechanisms must be provided. These may be point-to-point, bus-oriented, or network-oriented mechanisms and may involve a variety of protocols. To be flexible and have a wide range of applicability, software generators must be able to accommodate many communication mechanisms.

A.23 DYNAMIC MEMORY ALLOCATION

Dynamic memory allocation is not used in the implementation of navigation, guidance, and control systems software, so it is important that software generators not incorporate dynamic memory allocation in their capabilities.

A.24 PROPRIETARY LIMITATIONS

Simply using proprietary software is no problem. However, if proprietary software is modified to make it more useful, proprietary limitations may dictate the use of the modified software. In any case, licenses must be obtained to use proprietary software in both the development and postdevelopment support phase for all software.

Navigation, guidance, and control software that is proprietary should be protected under copyrights and patents because of the value of the software to the producer.

A.25 DATABASE ATTRIBUTES

There are many database attributes which are desirable, if not essential, in the design and implementation of navigation, guidance, and control software. Some of these are:

a. Consistency of databases.
b. Application-specific versus general databases.
c. Databases which support objects and object management.
d. Support of different schemata.
e. Support of different viewpoints.
f. Powerful default mechanisms in databases.
g. Configuration management and control.
h. Completeness.
i. Integration of tools with databases.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's Reference	3. Further Reference	4. Security Classification of Document
	AGARD-AR-292	ISBN 92-835-0663-4	UNCLASSIFIED

5. Originator	Advisory Group for Aerospace Research and Development North Atlantic Treaty Organization 7 rue Ancelle, 92200 Neuilly sur Seine, France
6. Title	AUTOMATED SOFTWARE GENERATION APPROACHES FOR THE DESIGN AND DEVELOPMENT OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE
7. Presented at	

8. Author(s)/Editor(s)	9. Date
Edited by Dr E.B.Stear and Prof. J.T.Shepherd	March 1992

10. Author's/Editor's Address	11. Pages
See Flyleaf	186

12. Distribution Statement This document is distributed in accordance with AGARD policies and regulations, which are outlined on the back covers of all AGARD publications.

13. Keywords/Descriptors

Guidance and control	Expert systems
Air navigation	Computer programs
Software support environments	Programming languages
Software engineering	Formal methods

14. Abstract

This Advisory Report summarises the findings and conclusions of Working Group 10 of the Guidance and Control Panel of AGARD, the Terms of Reference which were:

(i) To develop and consider a set of requirements for application software generators for guidance and control systems.

(ii) To evaluate the characteristics and capabilities offered by existing application software generator technology with respect to the requirements defined in (i).

(iii) If required, to determine the modifications and improvements necessary for such technology to meet the requirements defined in (i).

The Working Group defined and investigated four approaches to software generation: reusable software modules, expert systems, programme transformation techniques and fourth generation languages.

(i) To develop and consider a set of requirements for application software generators for guidance and control systems.

(ii) To evaluate the characteristics and capabilities offered by existing applications software generator technology with respect to the requirements defined in (i).

(iii) If required, to determine the modifications and improvements necessary for such technology to meet the requirements defined in (i).

The Working Group defined and investigated four approaches to software generation: reusable software modules, expert systems, programme transformation techniques and fourth generation languages.

ISBN 92-835-0663-4

(i) To develop and consider a set of requirements for application software generators for guidance and control systems.

(ii) To evaluate the characteristics and capabilities offered by existing applications software generator technology with respect to the requirements defined in (i).

(iii) If required, to determine the modifications and improvements necessary for such technology to meet the requirements defined in (i).

The Working Group defined and investigated four approaches to software generation: reusable software modules, expert systems, programme transformation techniques and fourth generation languages.

ISBN 92-835-0663-4

(i) To develop and consider a set of requirements for application software generators for guidance and control systems.

(ii) To evaluate the characteristics and capabilities offered by existing applications software generator technology with respect to the requirements defined in (i).

(iii) If required, to determine the modifications and improvements necessary for such technology to meet the requirements defined in (i).

The Working Group defined and investigated four approaches to software generation: reusable software modules, expert systems, programme transformation techniques and fourth generation languages.

ISBN 92-835-0663-4

(i) To develop and consider a set of requirements for application software generators for guidance and control systems.

(ii) To evaluate the characteristics and capabilities offered by existing applications software generator technology with respect to the requirements defined in (i).

(iii) If required, to determine the modifications and improvements necessary for such technology to meet the requirements defined in (i).

The Working Group defined and investigated four approaches to software generation: reusable software modules, expert systems, programme transformation techniques and fourth generation languages.

ISBN 92-835-0663-4

AGARD Advisory Report 292
Advisory Group for Aerospace Research and Development, NATO
AUTOMATED SOFTWARE GENERATION APPROACHES FOR THE DESIGN AND DEVELOPMENT OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE
Edited by E.B. Stear and J.T. Shepherd
Published March 1992
186 pages

This Advisory Report summarises the findings and conclusions of Working Group 10 of the Guidance and Control Panel of AGARD, the Terms of Reference which were:

P.T.O.

AGARD-AR-292

Guidance and control
Air navigation
Software support environments
Software engineering
Expert systems
Computer programs
Programming languages
Formal methods

AGARD Advisory Report 292
Advisory Group for Aerospace Research and Development, NATO
AUTOMATED SOFTWARE GENERATION APPROACHES FOR THE DESIGN AND DEVELOPMENT OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE
Edited by E.B. Stear and J.T. Shepherd
Published March 1992
186 pages

This Advisory Report summarises the findings and conclusions of Working Group 10 of the Guidance and Control Panel of AGARD, the Terms of Reference which were:

P.T.O.

AGARD-AR-292

Guidance and control
Air navigation
Software support environments
Software engineering
Expert systems
Computer programs
Programming languages
Formal methods

AGARD Advisory Report 292
Advisory Group for Aerospace Research and Development, NATO
AUTOMATED SOFTWARE GENERATION APPROACHES FOR THE DESIGN AND DEVELOPMENT OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE
Edited by E.B. Stear and J.T. Shepherd
Published March 1992
186 pages

This Advisory Report summarises the findings and conclusions of Working Group 10 of the Guidance and Control Panel of AGARD, the Terms of Reference which were:

P.T.O.

AGARD-AR-292

Guidance and control
Air navigation
Software support environments
Software engineering
Expert systems
Computer programs
Programming languages
Formal methods

AGARD Advisory Report 292
Advisory Group for Aerospace Research and Development, NATO
AUTOMATED SOFTWARE GENERATION APPROACHES FOR THE DESIGN AND DEVELOPMENT OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE
Edited by E.B. Stear and J.T. Shepherd
Published March 1992
186 pages

This Advisory Report summarises the findings and conclusions of Working Group 10 of the Guidance and Control Panel of AGARD, the Terms of Reference which were:

P.T.O.

AGARD-AR-292

Guidance and control
Air navigation
Software support environments
Software engineering
Expert systems
Computer programs
Programming languages
Formal methods

AGARD

NATO - OTAN

7 RUE ANCELLE · 92200 NEUILLY-SUR-SEINE

FRANCE

Téléphone (1)47.38.57.00 · Télex 610 176
Télécopie (1)47.38.57.99

DIFFUSION DES PUBLICATIONS
AGARD NON CLASSIFIEES

L'AGARD ne détient pas de stocks de ses publications, dans un but de distribution générale à l'adresse ci-dessus. La diffusion initiale des publications de l'AGARD est effectuée auprès des pays membres de cette organisation par l'intermédiaire des Centres Nationaux de Distribution suivants. A l'exception des Etats-Unis, ces centres disposent parfois d'exemplaires additionnels; dans les cas contraire, on peut se procurer ces exemplaires sous forme de microfiches ou de microcopies auprès des Agences de Vente dont la liste suite.

CENTRES DE DIFFUSION NATIONAUX

ALLEMAGNE
Fachinformationszentrum,
Karlsruhe
D-7514 Eggenstein-Leopoldshafen 2

BELGIQUE
Coordonnateur AGARD-VSL
Etat-Major de la Force Aérienne
Quartier Reine Elisabeth
Rue d'Evere, 1140 Bruxelles

CANADA
Directeur du Service des Renseignements Scientifiques
Ministère de la Défense Nationale
Ottawa, Ontario K1A 0K2

DANEMARK
Danish Defence Research Board
Ved Idraetsparken 4
2100 Copenhagen Ø

ESPAGNE
INTA (AGARD Publications)
Pintor Rosales 34
28008 Madrid

ETATS-UNIS
National Aeronautics and Space Administration
Langley Research Center
M/S 180
Hampton, Virginia 23665

FRANCE
O.N.E.R.A. (Direction)
29, Avenue de la Division Leclerc
92322 Châtillon Cedex

GRECE
Hellenic Air Force
Air War College
Scientific and Technical Library
Dekelia Air Force Base
Dekelia, Athens TGA 1010

ISLANDE
Director of Aviation
c/o Flugrad
Reykjavik

ITALIE
Aeronautica Militare
Ufficio del Delegato Nazionale all'AGARD
Aeroporto Pratica di Mare
00040 Pomezia (Roma)

LUXEMBOURG
Voir Belgique

NORVEGE
Norwegian Defence Research Establishment
Attn: Biblioteket
P.O. Box 25
N-2007 Kjeller

PAYS-BAS
Netherlands Delegation to AGARD
National Aerospace Laboratory NLR
Kluyverweg 1
2629 HS Delft

PORTUGAL
Portuguese National Coordinator to AGARD
Gabinete de Estudos e Programas
CLAFA
Base de Alfragide
Alfragide
2700 Amadora

ROYAUME UNI
Defence Research Information Centre
Kentigern House
65 Brown Street
Glasgow G2 8EX

TURQUIE
Milli Savunma Başkanlığı (MSB)
ARGE Daire Başkanlığı (ARGE)
Ankara

LE CENTRE NATIONAL DE DISTRIBUTION DES ETATS-UNIS (NASA) NE DETIENT PAS DE STOCKS DES PUBLICATIONS AGARD ET LES DEMANDES D'EXEMPLAIRES DOIVENT ETRE ADRESSEES DIRECTEMENT AU SERVICE NATIONAL TECHNIQUE DE L'INFORMATION (NTIS) DONT L'ADRESSE SUIT.

AGENCES DE VENTE

National Technical Information Service
(NTIS)
5285 Port Royal Road
Springfield, Virginia 22161
Etats-Unis

ESA/Information Retrieval Service
European Space Agency
10, rue Mario Nikis
75015 Paris
France

The British Library
Document Supply Division
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
Royaume Uni

Les demandes de microfiches ou de photocopies de documents AGARD (y compris les demandes faites auprès du NTIS) doivent comporter la dénomination AGARD, ainsi que le numéro de série de l'AGARD (par exemple AGARD-AG-315). Des informations analogues, telles que le titre et la date de publication sont souhaitables. Veuiller noter qu'il y a lieu de spécifier AGARD-R-nnn et AGARD-AR-nnn lors de la commande de rapports AGARD et des rapports consultatifs AGARD respectivement. Des références bibliographiques complètes ainsi que des résumés des publications AGARD figurent dans les journaux suivants:

Scientifique and Technical Aerospace Reports (STAR)
publié par la NASA Scientific and Technical
Information Division
NASA Headquarters (NTT)
Washington D.C. 20546
Etats-Unis

Government Reports Announcements and Index (GRA&I)
publié par le National Technical Information Service
Springfield
Virginia 22161
Etats-Unis
(accessible egalement en mode interactif dans la base de données bibliographiques en ligne du NTIS, et sur CD-ROM)

Imprimé par Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

SPS

Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

ISBN 92-835-0663-4